Titre: Title:	Static Probabilistic Timing Analysis for Real-Time Embedded Systems in Presence of Faults
Auteur: Author:	Chao Chen
Date:	2017
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Chen, C. (2017). Static Probabilistic Timing Analysis for Real-Time Embedded Systems in Presence of Faults [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2686/

UNIVERSITÉ DE MONTRÉAL

STATIC PROBABILISTIC TIMING ANALYSIS FOR REAL-TIME EMBEDDED SYSTEMS IN PRESENCE OF FAULTS

CHAO CHEN
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
JUILLET 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

STATIC PROBABILISTIC TIMING ANALYSIS FOR REAL-TIME EMBEDDED SYSTEMS IN PRESENCE OF FAULTS

présentée par: CHEN Chao
en vue de l'obtention du diplôme de: Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de:

M. KHOMH Foutse, Ph. D., président
M. BELTRAME Giovanni, Ph. D., membre et directeur de recherche
M. SAVARIA Yvon, Ph. D., membre
M. ZILIC Zeljko, Ph. D., membre externe

DEDICATION

To my family

ACKNOWLEDGEMENTS

My research work would not have been possible without the help of various people and grant-awarding organizations.

First of all, I am deeply grateful to my supervisor, Giovanni Beltrame, for his guidance, patience and encouragements. It has been a great pleasure to work with him and I have learned a lot from his knowledgeableness and thoroughness. He let me work independently on my research, while constantly providing invaluable advice and suggestions which led me to many new ideas. My Ph.D. program has been fruitful and Giovanni is no doubt the key to that.

I would like to thank MIST Lab members: Jacopo, Imane, Hassan and Mohamed. We had countless discussions on various ideas in different research fields. The discussions were fruitful and we consequently co-authored several papers together. Besides, they helped me a lot during the dissertation writing.

I wish to thank Luca and Jerôme. Thanks for inviting me to Toulouse and making my life joyful and fruitful with your organizations in France. The discussions inspired me a lot and I learned to do my research work from different perspectives.

Special thanks to other committee members: Foutse Khomh, Zeljko Zilic and Yvon Savaria.

I also wish to thank ReSMiQ for funding my research, which helped me focus on my research. It was my honor and I am grateful.

Finally, I am extremely thankful to my father, my mother, my brother and my sister. Their unconditional understanding and love were with me throughout my Ph.D. studies.

RÉSUMÉ

Une mémoire cache est le lien entre le processeur et la mémoire principale. Elle permet de réduire considérablement les temps d'accès aux blocs de mémoire dans un système embarqué temps-réel et critique (CRTES), ce qui influence énormément son comportement temporel. Des caches à accès aléatoire—caches avec une politique de remplacement aléatoire—ont été proposées dans le but d'améliorer les estimations du comportement temporel des CRTES, et cela en diminuant les cas pathologiques. Les Measurement Based Probabilistic Timing Analysis (MBPTA) et Static Probabilistic Timing Analysis (SPTA) sont deux méthodes qui ciblent à estimer le pire temps d'exécution (Worst Case Execution Time probabiliste - pWCET) d'une façon probabiliste et sécuritaire pour les caches aléatoires. À travers cette dissertation, on présente des travaux de recherche concernant l'estimation temporelle basée sur la méthode SPTA. L'état de l'art sur les méthodologies SPTA fournissent des estimations sécuritaires et strictes. En revanche, au vu de la réduction d'échelle des technologies des semi-conducteurs utilisés pour la mise en oeuvre des composants faisant partie des CRETS, les caches sur puce sont de plus en plus prédisposés aux pannes. Par conséquent, nous avons développé des méthodologies SPTA pour l'estimation des pWCETs en présence de pannes. Nous avons effectué également des évaluations de l'impact de ces fautes sur les comportements temporels.

Afin d'examiner les pannes, nous avons modélisé dans un premier temps les pannes transitoires et permanentes. Une panne transitoire représente un changement d'état temporaire. Le système peut ainsi être restauré en utilisant des techniques de détection et de correction des pannes. D'un autre côté, une panne permanente introduit un changement permanent. Elle persiste après son apparition et affecte en conséquence le comportement général du système. Nous avons alors proposé une méthode basée sur les chaînes de Markov afin de modéliser les états de disposition de la mémoire. Pour chaque accès à un bloc de mémoire, le changement de l'état est calculé en utilisant une matrice de transition, tout en tenant compte des impacts des fautes transitoires. Nous avons également utilisé différents types de modèles de la chaîne de Markov pour représenter le système ayant subi un nombres différent de pannes permanentes. Les expériences montrent que notre méthode SPTA assure des résultats précis en présence des pannes transitoires et permanentes.

Par la suite, nous avons étudié différents mécanismes de détection en ligne des pannes permanentes. Ce type de pannes joue un rôle important dans l'estimation temporelle, vu qu'elles continuent d'exister dans le système après leur production, ce qui affecte les accès en mémoire

qui suivent. Nous avons appliqué deux techniques de détection de pannes ; une technique de détection basée sur des règles, et une seconde technique basée sur le modèle Dynamic Hidden Markov Model (D-HMM). La détection basée sur les règles consiste à calculer le nombre de pannes pour chaque bloc de cache : si le nombre dépasse un certain seuil, le bloc de cache est considéré définitivement défectueux. La détection basée sur la D-HMM calcule la probabilité d'une panne permanente. Si cette probabilité dépasse un certain seuil, le bloc de cache est alors définitivement défectueux. Une fois que le bloc de cache est détecté comme définitivement défectueux, il n'est plus disponible pour les futurs accès. Nous avons découvert que les mécanismes de détection des fautes permanentes ont permis de changer considérablement les pWCETs d'un système. En effet, la détection basée sur la methode D-HMM a contribué énormément à l'amélioration du pWCET, comparé aux techniques de détection basée sur des règles.

Nous avons ensuite élargi le champ de nos travaux de recherche en ce qui concerne l'état de l'art sur la SPTA. Les travaux sur la SPTA sans doute peuvent être décomposés en deux parties : les méthodes basées sur l'énumération d'état et celles basées sur la contention. La première méthode est relativement similaire à notre méthode basée sur la chaîne de Markov, puisque les deux méthodes dépendent de l'incorporation des états dans le calcul du pWCET. Par conséquent, nous avons adapté notre méthode basée sur l'espace des états aux pannes avec la méthode basée sur l'énumération. Ensuite, nous avons renforcé la méthode basée sur la contention pour inclure l'impact causé par les pannes. Nous avons pris comme référence un mécanisme de détection des pannes dit parfait, ayant la capacité de détecter une panne permanente immédiatement après sa production. Finalement, nous avons comparé les pWCETs obtenus en utilisant la méthode de détection basée sur des règles et D-HMM avec ceux obtenus en utilisant la méthode de détection parfaite. Les mesures ont démontré qu'en moyenne, la D-HMM est plus susceptible à produire des pWCETs proches de l'idéal.

Dans le but d'analyser l'implémentation d'une détection des pannes permanentes, nous avons adopté une méthode de détection basée sur des règles avec un scrubbing périodique. Deux modes de commutation périodiques ont été choisis pour cet objectif : un mode auquel la détection de panne est active et un autre où celle-ci est ignorée. Pour chacun des deux modes, nous avons développé les formules correspondantes à la méthode basée sur l'énumération d'état et à la méthode basée sur la contention. Des évaluations expérimentales ont montré que notre méthode SPTA fournit un pWCET sécuritaire même dans le cas où le système contient un nombre limité de blocs de mémoire analysés avec la méthode basée sur l'énumération d'états. Ces évaluations démontrent également une amélioration de la précision qui va de pair avec l'augmentation du nombre de blocs de mémoire.

Finalement, nous avons comparé notre propre méthode basée sur la chaîne de Markov avec l'état de l'art sur la SPTA. Nous avons formellement présenté le cadre de travail du modèle de la chaîne de Markov et nous avons montré le niveau de sécurité des changements adaptatifs introduits dans le modèle de la chaîne de Markov. Les résultats expérimentaux indiquent que les deux méthodes sont caractérisées par des temps de calculs similaires, alors que le modèle basé sur les chaînes de Markov produit, en moyenne, des pWCETs plus précis. Les caches LRU (Least Recently Used) ont également été comparés aux caches aléatoires. Celles-ci ont montré que quand il s'agit d'un nombre limité de blocs de mémoire, les systèmes avec des caches LRU prennent moins de temps pour compléter l'exécution. Plus le nombre de blocs de mémoire s'accroit, plus les systèmes avec des caches aléatoires deviennent susceptibles à terminer plus tôt que ceux avec les caches LRU.

ABSTRACT

A cache is typically the bridge between a processor and its main memory. It significantly reduces the access latencies to memory blocks and its timing behavior. Random caches—caches with a random replacement policy—have been proposed to improve timing behavior estimates in critical real-time embedded systems (CRTESs) by reducing pathological cases due to systematic cache misses. Measurement Based Probabilistic Timing Analysis (MBPTA) and Static Probabilistic Timing Analysis (SPTA) aim at providing safe probabilistic Worst Case Execution Time (pWCET) estimates for random caches. In this dissertation, we present research work on timing estimation based on SPTA. State-of-the-art SPTA methodologies produce safe and tight pWCET estimates. However, as semiconductor technology scales down, CRTES components—especially their on-chip caches—become prone to faults. Consequently, we developed SPTA methodologies to estimate pWCETs in the presence of faults, and evaluated the impacts of faults on timing behaviors.

To investigate faults, we first defined transient and permanent fault models. A transient fault represents a temporary change of state. The system with transient faults can be recovered using fault detection and correction techniques. A permanent fault represents a permanent change of state. It persists after its occurrence and affects the system's behavior afterwards. Additionally, we proposed a Markov chain method to model memory layout states. For each memory block access, the state changes are calculated using a transition matrix. The transient fault impacts were integrated into the transition matrix computation, and we used different groups of Markov chain models to represent the system with different number of permanent faults. Experiments showed that our SPTA method provided accurate results in the presence of both transient and permanent faults.

Next, we studied different online fault detection mechanisms for permanent faults. Permanent faults play an important role in the timing estimates, because they exist in the system after their occurrences, and all following accesses are affected. We applied two permanent fault detection techniques, i.e., rule-based detection and Dynamic Hidden Markov Model (D-HMM) based detection. Rule-based detection counts the number of faults for each cache blocks: if the number exceeded a threshold, the cache block is regarded as permanently faulty. D-HMM based detection predicts the probability of a permanent fault. When the prediction is larger than a threshold, the cache block is classified as permanently faulty. Once a cache block is detected as permanently faulty, it is disabled for future accesses. We found that permanent fault detection mechanisms drastically changed the pWCETs of a system, and

D-HMM based detection produced an improved pWCET compared to rule-based detection.

We then extended the scope of our research towards state-of-the-art SPTA. The state-of-the-art, non-fault aware SPTAs consist of two parts: a state enumeration-based method and a contention-based method. The state enumeration-based method resembles our Markov chain based method, in that both methods adopt states for pWCET calculations. Consequently, we adapted our state-space based method with faults within the state enumeration-based method. Then, we extended the contention-based method to account for fault impacts. As a reference, we adopted a perfect permanent fault detection mechanism that detects a permanent fault immediately after it happens. We compared pWCETs obtained using rule-based and D-HMM based detection to those obtained while using a perfect detection. Measurements demonstrated that, on average, D-HMM could produce pWCETs that were closer to the ideal.

For the analysis of the effects of the implementation of permanent fault detection, we adopted rule-based detection with periodic scrubbing. We used two periodic switching modes: a mode in which fault detection is active and one in which it is ignored. For each of the two modes, we developed the corresponding formulae for the state enumeration-based method and the contention-based method, respectively. Experimental evaluations showed that our SPTA method provided a safe pWCET result even with few memory blocks and its accuracy improved when the number of memory blocks increased.

Finally, we compared our own Markov chain based method to this state-of-the-art SPTA approach. We formally presented the Markov chain model framework and explained the safety of the adaptive changes in the Markov chain model. Experimental results indicated that both methods had similar calculation time, while the Markov chain model produced, on average, more accurate pWCETs. Least Recently Used (LRU) caches were also compared to random caches, which showed that when few memory blocks were used, systems with LRU caches took less time to complete their execution. As more memory blocks are used, systems with random caches might terminate earlier than those with LRU caches.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

1-CDF	inverse Cumulative Distribution Function
BM	Block Maxima
CFG	Control Flow Graph
CRTES	Critical Real-Time Embedded System
DCDR	Detection, Correction, Diagnosis and Reconfiguration
DTM	Degraded Test Mode
DVS	Dynamic Voltage Scaling
D-HMM	Dynamic Hidden Markov Model
ECC	Error Correcting Code
ETM	Execution Time Model
ETP	Execution Time Profile
EVT	Extreme Value Theory
FIFO	First In First Out
FMM	Fault Miss Map
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
HCI	Hot Carrier Injection
HEO	Highly Elliptical Orbit
i.i.d.	independent and identically distributed
ILP	Integer Linear Programming
LRU	Least Recently Used
MBPTA	Measurement Based Probabilistic Timing Analysis
MPSoC	Multi-Processor System-on-Chip
MRU	Most Recently Used
MTTF	Mean Time To Failure
MWC	Multiply With Carry
pWCET	probabilistic Worst Case Execution Time
POT	Peaks Over Threshold
PRNG	Pseudo-Random Number Generator
PTA	Probabilistic Timing Analysis
PUB	Path Upper Bounding
RW	Reliable Way
SEC-DED	Single Error Correction-Double Error Detection

SEU	Single Event Upset
SPTA	Static Probabilistic Timing Analysis
SRB	Shared Reliable Buffer
WCET	Worst Case Execution Time
WCRT	Worst Case Response Time

CHAPTER 1 INTRODUCTION

Recent years have witnessed an unprecedented growth in the number of critical real-time embedded systems (CRTESs). Airplanes, self-driving cars, and spacecraft are examples of such systems, in which the timing of a computation is as important as its result, and in which a failure could result in loss of equipment or life; Imagine an aircraft's response time to a pilot input, or the activation of a collision avoidance system in an autonomous vehicle. With the continued trend in automation, the demand for CRTES is expected to continue growing in the future.

Since timing is such an crucial aspect of these systems, many techniques have been developed to verify this critical non-functional property. Among these, timing analysis is one of the most used technique to ensure the safe operation of CRTES [116, 73, 74].

Timing analysis strives to provide guarantees for the maximum time needed to perform a given computation, providing its safe Worst Case Execution Time (WCET). However, one of the main obstacles to accurate timing analysis is the unpredictable timing behavior of modern computer architectures; with multi-stage pipelines and multi-level memory hierarchies, it is extremely difficult to accurately predict the execution time of a given program. This is made almost impossible with parallel architectures (such as multi-core systems), due to the presence of shared resources. Being conservative and over-estimating does not solve the problem, because of the extremely wide gap between the worst and average cases [18].

In this context, a probabilistic analysis approach can be beneficial [18]: by enabling true randomized behavior in all the components of a computer, one can define probabilistic metrics to the timing behavior of a system. Successful implementation of such systems will have tremendous impact on the way critical systems are designed. The potential benefits in terms of cost of integration, verification, and certification of real-time software are enormous. For that matter, consider the development of the on-board computer of a satellite. This is an excellent example of a CRTES: if commands or alarms are not treated in the appropriate time frame, the entire satellite could be lost. A traditional approach would take a flight-proven processor (e.g. the LEON3) equipped with a real-time operating system (e.g. RTEMS [35]) and statically schedule all software tasks to guarantee that the control software will always respond to events within a safe time frame. This requires detailed knowledge of the hardware and expensive software analysis. The behavior of multiple interacting tasks can lead to rare *corner cases*[1] that can lead to catastrophic effects. The largest part of the cost of software

[1] A corner case is a problem or situation that occurs only outside of normal operating parameters

qualification is to identify and deal with these situations. The base idea with probabilistic systems is that instead of struggling to eradicate corner cases, one can tweak the system parameters to reduce their probability of occurrence to negligible values.

As a result, Probabilistic Timing Analysis (PTA) methods have been proposed to provide the upper bound of randomized systems. Unlike a single WCET value derived from a deterministic system, the timing distribution of a random system behavior is a probabilistic WCET (pWCET), i.e. the probabilities of execution times exceeding given values. Three types of PTA methods were proposed in the literature: Measurement Based Probabilistic Timing Analysis (MBPTA) [37], Static Probabilistic Timing Analysis (SPTA) [118, 27, 39, 9] and a hybrid of both MBPTA and SPTA [18]. The trustworthiness of these three approaches is investigated in [6], which concludes that PTA methods have promising pWCET distributions, but more industrial support is needed to promote their use in real-world products.

That being said, most of the existing PTA methods are typically not fault aware. A few years back, this would have been acceptable. However, as technology advances, the continuous shrinking of feature sizes has made reliability a challenge that cannot be overlooked [12]. The system needs to meet real-time requirements while dealing with reliability issues. The presence of faults, for instance, may degrade system performance and cause catastrophic consequences. As a matter of fact, faults account for 80 percent of unmanned aerial vehicles failures reported in a reliability study of the US Office of the Secretary of Defense [95]. Aerospace systems especially navigate in particularly harsh environment in which high-energy particles interfere with the systems and cause different types of faults. This renders it one of the most challenging problems for space applications.

Researchers have performed numerous studies on faults of semiconductor devices to help understand their impacts. Parameter and dynamic variations and their impact on circuit failures, caches in particular, have been studied [24, 25]. Nassif *et al.* [82] demonstrate that SRAM—an element often used in caches—will be affected significantly by technology scaling. Hardy and Puaut [56] report that caches will be a potential source of performance degradation in future designs. Consequently, we can see that faults affect system performance drastically, making it vital to develop PTA methods that take into account fault impacts, an issue that will be principally addressed in this dissertation.

On that same note, to improve the timing behaviors of CRTES and reduce the cost of timing analysis, different architectures and analysis methodologies have been proposed in the literature. One key architecture component is the cache of a processor. A cache is a fast and expensive memory that is physically close to the processor, which provides instructions and data at a rate comparable to the computing speed of a processor. Hence the memory

access time can be significantly reduced if the required information is stored in the cache. When timing analysis is of concern and in the case of a cache miss, the number of cycles to execute an instruction is much larger than that in the case of a cache hit. Therefore, there may be significant timing variations for the execution of the same instruction when performing timing analysis, depending on the context. To obtain safe results, a conservative timing analysis is performed. One needs to find the pathological cases—those that lead to systematic cache misses—of all possibilities, which requires time and effort. Pathological cases may be much worse than the average case due to systematic cache misses by a particular pattern, and it is extremely difficult to trigger since many factors may produce it, such as memory layout, interaction with other programs, etc. In this dissertation, we study a random cache architecture with a random replacement policy. It reduces the dependence on a program's execution history by randomly evicting cache blocks, thus minimizing the impact of pathological cases and easing the timing analysis [90].

Timing analysis for random caches, however, is now facing new challenges. In this dissertation, we develop new SPTA methodologies for random caches, with the ability to take faults and their corresponding detection and correction techniques into consideration. Using our methodologies, we consider fault impact on the system design phase to manage faults effectively, which is critical for CRTES, especially for aerospace systems that experience harsh low pressure and high radiation environments[81].

1.1 Basic Concepts and Definitions

Pathological Cases

Caches play a fundamental role in determining the execution time of software. A cache hit provides a much faster access to data and instructions compared to accessing the main memory. The cache hit and miss pattern is affected by the size of a program's code and data, memory addresses, access history, interrupts and preemptions [19]. Pathological cases refer to the design patterns that cause significant cache misses and make Worst Case Execution Time (WCET) very pessimistic.

As shown in [19], a good memory layout produces all cache hits after the code is stored in the cache. However, a bad memory layout may induce a pathological case in which a 100% of cache misses happen, which dramatically increases the execution time and is very difficult to predict or test. However, using a random cache, the probability of a pathological case can be extremely low compared to traditional deterministic caches [90], which drastically improves the worst-case performance of the system.

Random Replacement

Fully-associative or set-associative caches have several cache blocks in each cache set. When a cache set is fully occupied, the replacement policy determines which cache block to evict and replace. For example, a Least Recently Used (LRU) replacement policy finds the least recently used cache block and replaces it with the new cache block. Intuitively, a random replacement chooses a cache block randomly whenever it is needed.

Two random replacement policies exist: evict-on-miss replacement [27] and evict-on-access replacement [39]. The evict-on-miss replacement policy evicts a cache block randomly when a cache miss happens, and puts the incoming cache block into the evicted cache block. The evict-on-access replacement policy evicts a cache block randomly when a cache is accessed by the processor, whether it is a cache hit or a cache miss. If a cache miss happens, the new cache block replaces the evicted one.

In this dissertation, we study SPTA using the evict-on-miss replacement policy since it has been reported to provide a better performance compared to an evict-on-access replacement policy [39]. Thus, evict-on-miss replacement policy is hereinafter referred to as simply a random replacement policy unless stated otherwise.

Execution Time Profile

An Execution Time Profile (ETP) is often used in timing analysis (PTA) [18, 20, 37, 27] in which it represents the frequencies of the possible execution times of software. It consists of two parts: the execution times (in terms of cycles or number of cache misses) and their corresponding probabilities.

An ETP can be represented as $ETP = \{T, P\}$, where T denotes execution times, i.e., $T = [t_0, t_1, t_2, ...]$, and P indicates the corresponding occurrence probabilities for each execution time, i.e., $P = [p_0, p_1, p_2, ...]$, with p_i being the occurrence probability of execution time t_i.

Probabilistic WCET

For traditional deterministic systems, deterministic timing analysis techniques produce a single WCET estimate, which upper-bounds the system execution time, and numerous techniques have been developed to improve the tightness of the WCET. For random systems, PTA techniques are used to compute multiple WCET estimates with corresponding probabilities, instead of a single upper bound estimate. The final result is denoted as a probabilistic WCET (pWCET).

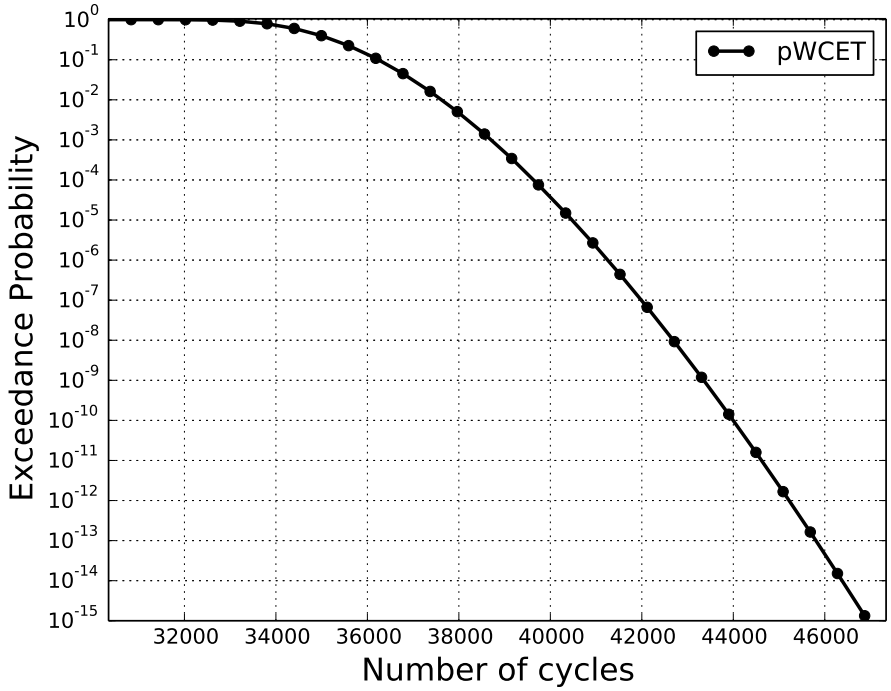


Figure 1.1 An example of pWCET.

Figure 1.1 shows an example of a pWCET. The x-axis denotes the execution time, and the y-axis indicates the exceedance probability, i.e. the probability to exceed a given execution time. The exceedance probability is the inverse Cumulative Distribution Function (1-CDF), i.e. the complementary CDF.

The CDF can be calculated by integrating an ETP and the sum of CDF and 1-CDF is always 1. Therefore, the pWCET can be represented using two variables T and P_{1-cdf}. $T = [t_0, t_1, t_2, ...]$ is the timing vector from the ETP and $P_{1-cdf} = [p^0_{1-cdf}, p^1_{1-cdf}, p^2_{1-cdf}, ...]$ is the corresponding exceedance probability vector. Thus, we have $p^i_{1-cdf} = \sum_{\forall j, t_j > t_i} p_j$, where p_j is from the probability vector P in ETP. With the help of pWCET, we can observe if the probability of the timing failure of the system meets the safety or operational requirements, e.g., 10^{-9} per hour.

Measurement Based Probabilistic Timing Analysis

MBPTA is based on Extreme Value Theory (EVT) [37, 28], and collects measurements from the target hardware to obtain trustworthy pWCET estimates. EVT aims at estimating the probabilities of events that seldom occur (or even never occurred before), and is used to predict rare events in many different fields, such as financial crisis, flood levels, catastrophic

weather conditions, etc. The estimation procedures can be performed using two approaches: Block Maxima (BM) and Peaks Over Threshold (POT). The BM approach breaks the data into n groups, and uses the maximum data from each group. The POT approach exploits a threshold and all data that exceed the threshold are used for estimation.

Due to the presence of random caches, MBPTA uses execution time measurements that vary under the same condition as the data. With the help of EVT, MBPTA can compute pWCETs of any PTA-compliant platform, such as [59, 69, 87, 15], using the procedure described in the following. The measurements can be represented as independent and identically distributed (i.i.d.) variables. Then, they are grouped using the BM or POT approach. The distribution which fits the data is estimated, and the pWCET is calculated using the parameters of the distribution.

MBPTA has been used extensively in real-world applications [68, 94] and especially in critical real-time systems [111, 112]. It has proven to be a sound approach for estimating pWCET using measurements [37, 104].

Static Probabilistic Timing Analysis

SPTA requires detailed knowledge of a program and the target system for its timing analysis. An SPTA approach uses the memory traces or the compiled executable binary and calculates pWCET exploiting the additional provided information, such as cache associativity, reuse distances of memory blocks, etc. The computational complexity of an SPTA approach that aims at performing a complete analysis of the program increases exponentially with the number of memory blocks used. Therefore, some information may be discarded in the calculation process to reduce the computation complexity, while maintaining a good accuracy at the same time, as is described in [27, 9].

Various SPTA approaches have been tested using benchmarks. They can typically produce results with good accuracy and, as more information is used, the result quality can be improved at the cost of calculation speed.

Reuse Distance

The reuse distance is a metric often used in SPTA. It defines the number of memory block accesses between accesses to the same memory block. When the memory block appears for the first time, its reuse distance is defined as ∞. For example, assume to have a memory trace a, b, c, a, c, b, we can see that the reuse distances are $\infty, \infty, \infty, 3, 2, 4$ respectively. With knowledge of the reuse distance and cache associativity, simplified SPTA methods have been

developed which provide safe pWCET estimates [27, 39].

Transient Faults

As technology scales down, the probability of fault occurrences increase so dramatically that we cannot afford to ignore them any longer. In this dissertation, we consider Single Event Upsets (SEUs) as our main source of transient faults. These are caused by high energy particles, especially in high-radiation environment of avionics [105]. The internal state of the circuit is changed by the particles, and a restorable fault occurs as a result. However, SEUs do not damage the circuit permanently. Numerous error detection and correction techniques have been proposed, such as parity bit detection, turbo code correction, etc. Once the fault is detected and corrected, the fault can be removed from the circuit.

Permanent Faults

Technology scaling also makes the circuit less reliable and permanent faults may consequently occur as well. There are two types of permanent faults: faults due to variations in manufacturing process, and those that are caused by component wear-out effects, aging, etc. The former type of permanent faults is static, i.e. they remain the same after the manufacturing of the component and the number of faults do not change. The latter is dynamic, and additional permanent faults may occur after some time due to aging, etc. In this dissertation, our focus is on dynamic permanent faults. Once permanent faults happen, the circuit with such faults is permanently damaged and cannot be recovered.

1.2 Dissertation Objectives

The main objective of our research work is to develop SPTA methodologies for random caches in the presence of transient and dynamic permanent faults, taking existing fault detection and correction techniques into consideration. The main objective includes the following sub-objectives:

- Develop an SPTA methodology for random caches without faults. The SPTA methodology is used to provide safe and tight pWCET in absence of faults. We developed a Markov chain based SPTA method for instruction caches [31] and then extended it to write-back data caches [33]. Since prior SPTA methods exist, we compared our SPTA method to the state-of-the-art SPTA methods to make sure that our SPTA produces results with good accuracy at a reasonable computation cost [33].

- Develop an SPTA methodology for random caches, while taking transient and permanent faults into account. The presence of faults changes the timing behavior of the system. Transient and permanent faults affect the system differently and, for them, specific fault models are established. Fault detection and correction techniques are assumed to be already in place and their implementation details omitted. Starting from these fault models and the SPTA method without faults, we developed an SPTA methodology capable of accounting for fault impacts [31].

- Investigate the impact of permanent fault detection mechanism on timing behaviors. There are numerous ways in which permanent fault detection can classify a fault as permanent. Since a permanent fault cannot be recovered for future operations, its occurrence and detection has a big impact on timing analysis. We compared two different online permanent fault detection mechanisms and studied their impact on the timing behavior of the instruction cache of a system [30]. Write-through data caches, those that avoid data inconsistency issues due to faults, were applied for full performance analysis of detection techniques [32].

- Develop an SPTA methodology that deals with the implementation details of permanent fault detection mechanisms. In reality, a permanent fault detection mechanism cannot classify a fault as permanent immediately after its occurrence. We assume that a permanent fault detection is implemented using period scrubbing. We extended our SPTA method into two operating modes that affect each other, studying the mode with and without periodic detection separately [34].

1.3 Dissertation Organization

This dissertation is organized as follows:

- Chapter 2 reviews related work on timing analysis for random caches in three main areas: randomization implementation of a system; PTA methodologies for random caches; and fault occurrences due to technology scaling and external environment.

- Chapter 3 establishes transient and permanent fault models for random caches. A detection mechanism is used to detect a transient fault whenever it occurs in a cache block, and the content in this block is set as invalid. For the permanent fault, a perfect detection mechanism that detects and disables the cache block immediately after it occurs is used. We developed a state space based Markov chain model for SPTA, and

extended it to account for fault impacts, by calculating state changes at each memory block access.

- Chapter 4 studies two online permanent fault detection mechanisms, i.e., rule-based and Dynamic Hidden Markov Model (D-HMM) based detection. We assessed how different detection mechanisms affect pWECTs, and compared the effectiveness of both mechanisms.

- In Chapter 5, we applied our state space method to state-of-the-art SPTA, which is composed of a state enumeration method and a contention based method. We also integrated impact of faults into the contention based method for a simplified and fast SPTA. Furthermore, two detection mechanisms—rule-based and D-HMM based detection mechanisms—are compared to a perfect detection mechanism for the statistical analysis of fault detection impacts.

- The implementation of permanent fault detection is investigated in Chapter 6. We use a periodic scrubbing technique for fault detection. When a permanent fault is detected, the corresponding cache block is disabled for future use. We tackled the implementation impact by developing two SPTA formulae in two modes, and switching between them periodically.

- Chapter 7 introduces our framework for Markov chain based SPTA. We compared Markov chain based SPTA to state-of-the-art SPTA. Additionally, we evaluated performance of random caches and Least Recently Used (LRU) caches, respectively.

- Chapter 8 provides a general discussion on the proposed SPTA methodologies and the impact of faults and their detection techniques.

- We then conclude the dissertation and discuss future work in Chapter 9.

- Finally, we present two co-authored articles in the APPENDICES, which demonstrate the issues of a random cache implementation on a Field Programmable Gate Array (FPGA) and the impact of a random cache on scheduling policies for multicore systems, respectively.

CHAPTER 2 LITERATURE REVIEW

In the verification of a critical real-time embedded system (CRTES), the WCET plays an important role. The components on a CRTES can be classified into two categories: deterministic and stochastic. Deterministic components are adopted for traditional computer architectures, in which WCETs are computed to ensure that all tasks meet deadlines. Three types of timing analysis methods are developed [116] accordingly to provide trustworthy WCETs, i.e. static timing analysis [80, 106, 47, 110, 84, 115, 103, 91, 89, 54, 57, 29, 76], measurement-based timing analysis [88, 113, 114, 42] and hybrid timing analysis [18, 22, 43].

Aside from deterministic components, numerous stochastic components have been proposed to randomize the system, thus improving the system performance [100, 101, 90, 66]. Consequently, PTA methodologies are proposed to calculate pWCETs, i.e. execution times with respect to exceedance probabilities. In this chapter, we review the work on random systems in three orthogonal areas, i.e. randomization approaches, PTA methodologies for randomized systems, and the impact of faults on systems studied through PTA. Table 2.1 summarizes the work and their contributions at the end of the chapter.

2.1 Randomization

2.1.1 Cache Randomization

Many techniques have been proposed to modify caches so that they behave randomly. Schlansker *et al.* [96] present a design of set-associative data caches with a randomized placement policy. When the data is placed into the cache, a pseudo-random hash function randomly selects a cache set. A matrix multiplication program evaluation showed that a random placement policy can distribute data uniformly across different cache sets, hence avoiding the sharply peaked distribution of a modulo placement policy. Thus, data caches with a random placement policies provide a lower cache miss ratio even at a higher fill fraction.

Topham and Gonzalez [107] demonstrate how to modify the cache indexing function to reduce the cache conflict misses. They propose a polynomial modulus function for cache indexing. Using the SPEC95 benchmarks, they find that the cache miss ratio standard deviation is reduced from 18.49 to 5.16, thus increasing the predictability of the system.

Instead of modifying the placement policy, Quinones *et al.* [90] propose to modify the replacement policy for single-level instruction caches. When a cache block has to be evicted, is is randomly selected, instead of using a deterministic pattern (e.g., LRU). The authors validate

their replacement policy in standard and skew-associate caches, showing that it increases the average execution time but avoids pathological cases of systematic misses. The worst-case execution times are 33% and 25% faster for standard and skewed caches, respectively.

In addition to a random replacement policy, Kosmidis *et al.* [66] develop a parametric random placement, such that PTA can be applied to set associative as well as direct-mapped caches. The parametric hash function uses a seed to generate random indexes that are deterministic throughout the program execution. For different executions, the seed is modified so that the execution times can be represented as i.i.d. variables. Evaluations show that the hardware complexity and energy consumption for parametric random placement is low, and the average performance is comparable to that of deterministic caches. Another Pseudo-Random Number Generator (PRNG) is proposed in [7]. This meets the safety requirements of IEC-61508 SIL 3 [1] and can be used to provide randomization for multicore platforms.

Hernandez *et al.* [60] propose a random modulo cache design that uses randomized-permutations of memory address index bits. This makes the memory mapping and the layout independent, while taking advantage of spatial locality. Thus, pWCET is improved when compared to the random placement cache design in [66].

Reineke [92] state that it is harmful to use random caches in hard real-time systems, because timing analysis techniques for deterministic LRU caches are preferable to SPTA and MBPTA techniques. However, Mezzetti *et al.* [78] point out that MBPTA techniques can produce trustworthy pWCET estimates for random caches. New, improved SPTA techniques are still being developed. For example, in loops where the number of memory blocks exceeds the cache associativity, random caches outperform LRU caches that incur only in cache misses. Using state-of-the-art SPTA techniques [11], pWCET can be computed.

2.1.2 Bus Randomization

Besides caches, another hardware component that can be randomized is the bus. Jalle *et al.* [61] describe buses whose arbitration policies make execution times fulfill PTA requirements for multicore processors. They study a lottery arbitration bus and propose a randomized-permutation arbitration bus. On each arbitration round, the lottery arbitration bus randomly grants access to cores [70]. However, in this case, one core may take many rounds to access the bus. Therefore a randomized-permutation arbitration bus produces a random permutation for all cores, which sets an upper bound to the wait time for requesting cores. In fact, there is no need to develop extra timing analysis techniques, for the PTA techniques used for random caches with a single core processor can be directly applied to estimate pWCET of random buses.

2.1.3 Software Randomization

Some software techniques can be used to enforce randomness in a system as well. Berger *et al.* [16] introduce a memory manager—DieHard—to achieve probabilistic memory safety. DieHard is a runtime system that aims at probabilistically avoiding all memory errors. It approximates a heap of infinite size, in which it allocates objects randomly. To further enhance security, the object can be replicated and all replicas can run simultaneously, which helps to avoid errors from illegal reads. Evaluations over the full SPECint2000 suite [102] show that DieHard effectively reduces memory errors.

A runtime system and compilers are used to randomize program executions with traditional deterministic caches to fulfills PTA requirements [67]. Memory objects are randomly placed by the compiler and the runtime system during each execution, which effectively makes deterministic caches behave randomly. The randomization results are evaluated using Stabilizer [38]. Experiments show that the proposed technique generates execution times with i.i.d. property, which makes MBPTA suitable for pWCET estimates.

Kosmidis *et al.* [64] present a static software randomization approach, which randomly places functions, global variables and stack frames at compile time. An extra padding is created to help determine stack frame locations. Evaluations show that this approach provides affordable functional verification cost to fulfill safety requirements and is easy to implement at the cost some extra storage overhead.

2.2 Probabilistic Timing Analysis

In this dissertation, we focus on random caches for system randomization. As mentioned in previous sections, there are three types of PTA for random caches: MBPTA, SPTA and the hybrid methods jointly using MBPTA and SPTA. In this section, we separately review each one of the PTA methods.

2.2.1 Measurement Based Probabilistic Timing Analysis

MBPTA collects timing measurements on a target system, and derives pWCET using statistical approaches. Cucu-Grosjean *et al.* [37] propose an EVT based MBPTA approach for multi-path programs with given input vectors for systems with random caches. They first collect execution time measurements and then group the measurements using the BM approach that selects the maximum value in each group for processing. The Gumbel distribution parameters are estimated for fitting and the pWCET distribution is derived from

these parameters. Benedicte *et al.* [14] present a method that gives suggestions as to the choice of a sufficient number of measurements for MBPTA.

To extend MBPTA to any input vector, Kosmidis *et al.* [62] propose the Path Upper-Bounding (PUB) method, which probabilistically upper-bounds the execution time of any path. It upper-bounds core latency and cache latency, and performs code alignment to obtain a safe pWCET estimate. Ziccardi *et al.* [119] present a technique which modifies the time measurements to represent the characteristics of all paths and to make them probabilistically path independent. Path independent execution times are combined as synthetic measurements to account for unobserved paths.

2.2.2 Static Probabilistic Timing Analysis

Unlike MBPTA, SPTA statically analyzes the program to discover its timing behavior. Zhou [118] presents a cache hit probability calculation formula that provides the lower bound value using the reuse distance. However, Cazorla *et al.* and Altmeyer *et al.* [27, 9] find the pWCET calculation with this formula to be unsound, because the pWCET distribution is calculated using convolutions where all lower bound hit probabilities are considered independent. With limited cache associativity, this is not the case and therefore the pWCET result may be wrong.

To provide a safe pWCET estimate, Davis *et al.* [39] develop an independent cache hit probability formula which uses both the reuse distance and the cache associativity. When the reuse distance is larger than the cache associativity, the hit probability is 0. From this, the lower bound probabilities are computed for all memory blocks. An Execution Time Profile (ETP) can be derived using the cache hit probability of a memory block, and the final pWCET distribution is computed by convolutions of all ETPs, since they are independent.

Kosmidis *et al.* [66] propose another cache hit probability formula which replaces the reuse distance with the expected number of misses between two memory blocks as the exponent. Davis [40] has refuted the formula, and provided a case showing that it may be optimistic to derive pWCET using it.

To estimate pWCET more precisely, Altmeyer *et al.* [9] use a state enumeration method which considers all the possible states in a program's execution. For each memory access, the entire state information is calculated to produce an accurate result, however, the number of states increases exponentially with the number of memory blocks. To cope with this problem, a cache contention based method is developed to derive less precise pWCET results with faster calculation. By combining both the state enumeration and the cache contention based

methods, pWCET estimation becomes tractable and the result is a compromise between calculation accuracy and speed. Altmeyer *et al.* [11] extend their work by developing another heuristic to decide how to select the memory blocks for the state enumeration and the cache contention based method, respectively. Furthermore, they add an approach to calculate lower bound hit probabilities using the stack distance (an alternative formula for reuse distance).

Griffin *et al.* [49] explain a state space based approach, and reduce the state sizes using lossy compression techniques. May and Must analysis [80] are performed respectively, where the May analysis finds the memory accesses that are guaranteed to be cache misses and the Must analysis finds those that are guaranteed to be cache hits. With lossy compression, some memory blocks are replaced by unknown memory blocks, and the states and their timing history are replaced by bounding information, which results in two compression methods: hit probability and forward reuse distance. Benchmark evaluations indicate that both methods are tractable and derive precise pWCETs when using appropriate parameters.

Lesage *et al.* [71] analyze the Control Flow Graph (CFG) of a multi-path program to find the worst-case execution path using a joint function to explore cache states and path inclusions. They apply the state-of-the-art SPTA in [9] to the worst-case path, and derive the pWCET distribution. This multi-path pWCET estimation technique produces a more accurate result when compared to a previous work on multi-path pWCET derivation using simple path merging techniques [39].

2.2.3 Hybrid Probabilistic Timing Analysis

Bernat *et al.* [18] aim at reducing the overestimation of execution times and propose a hybrid method using both measurement and static approaches for their timing analysis. The program is represented as a syntax tree, and the ETPs of blocks are computed respectively. Taking block interactions into consideration, the final execution time is obtained using joint operators for ETPs. A pWCET calculation tool framework is developed in [20]. It generates instrumentation and traces, and traverses the syntax tree to perform a hybrid analysis.

2.3 Fault Impact

Due to technology scaling, fault rates have increased noticeably. As a result, PTA methods in the presence of faults have been investigated to study how faults affect the timing behaviors of randomized systems. For random caches, Slijepcevic *et al.* [98] propose the Degraded Test Mode (DTM) to derive safe pWCET distributions. DTM specifies requirements on fault-tolerant hardware that is compliant with PTA. Together with MBPTA, DTM provides safe

and tight pWCETs, and graceful degradation of the average and worst-case performance in the presence of permanent faults.

To further the study of fault scenarios, Slijepcevic *et al.* [99] considered both transient and permanent faults, in addition to error Detection, Correction, Diagnosis, and Reconfiguration (DCDR) techniques. Different fault rates are applied to a system with random caches. pWCET distributions from MBPTA indicate that—due to the presence of DCDR—pWCETs are negligibly affected. However, permanent faults affect the pWCET estimates significantly in some cases.

Besides fault-aware MBPTA, SPTA techniques have also been adapted to consider fault impact. Hardy and Puaut [56] propose an SPTA method to deal with fault impact in LRU instruction caches. The permanent faults that are caused by process variations in the manufacturing phase are taken into account. They first compute the fault-free timing behavior of the system, and then they generate a Fault Miss Map (FMM) using an Integer Linear Programming (ILP) solver with faulty blocks. By combining the fault-free timing behavior and the FMM, they obtain pWCET distributions in the presence of the permanent faults introduced by the manufacturing process.

Hardy *et al.* [55] introduce two reliability mechanisms—Reliable Way (RW) and Shared Reliable Buffer (SRB)—to mitigate the impact of permanent faults. RW adds one extra fault-free cache block to each cache set, while SRB adds a shared fault-free cache block that can be used by all cache sets, in addition to an extra look-up mechanism for this shared block in a cache. pWCETs are computed using SPTA techniques [56], and the results show that, with these reliability mechanisms, the pWCETs can be improved dramatically.

Trilla *et al.* [109] propose a new random modulo cache design with the typical features of a random modulo cache but also improved reliability. With the help of MBPTA, they assess the random modulo robustness against Hot Carrier Injection (HCI) aging, which shows that—compared to the random modulo cache in [60]—the lifetime of the new design increases by 3.9 times and 8.8 times for instruction cache and data cache, respectively. The critical path is not affected by the new design.

Table 2.1 Summary of related research work in the literature.

Feature	Reference	Contribution
Cache Randomization	[96, 107, 66, 60]	Random placement policies for caches using different strategies.
	[90]	Cache random replacement policy.
Bus Randomization	[61]	Bus random arbitration policy.
Software Randomization	[16, 67]	A compiler or a runtime environment that randomly allocates objects into the memory.
	[64]	Static approach to randomly place objects at compile time.
MBPTA	[37, 14]	EVT based approach to predict pWCET for given input.
	[62, 119]	Representation of all paths as the input to EVT based approach.
SPTA	[118, 27, 40, 39, 66]	Independent cache hit probability for each memory request.
	[9, 11, 49]	State space based approach for precise estimates.
	[39, 71]	Multi-path program analysis.
Hybrid PTA	[18, 20]	Hybrid analysis using syntax trees.
Fault Impact	[98, 99]	MBPTA in the presence of faults.
	[56, 55]	SPTA for LRU caches with manufacturing permanent faults.

CHAPTER 3 ARTICLE 1: STATIC PROBABILISTIC TIMING ANALYSIS IN PRESENCE OF FAULTS

3.1 Preface

Random caches are starting to be more commonly used in commercial processors to improve the system performance. Reliability is also becoming a serious challenge for semiconductor devices. Smaller feature sizes, lower power voltages and higher frequencies have increased the probabilities of faults in cache memories which adopt the smallest features allowed. In this article, we propose a model for transient faults—i.e., fault that affect cache blocks temporarily and does not hinder successive data storage operations—and one for permanent faults—i.e., faults that permanently damage cache blocks—for random caches. On top of that, we develop an SPTA technique that describes the states of the cache for each memory request and uses them to derive pWCET distributions, while taking the impact of faults into consideration. This is our first attempt to develop SPTA methodologies and to study fault impacts. Experiments prove that it is possible to deal with faults using SPTA techniques and the pWCET estimates could be as precise as those from simulations provided that we use sufficient memory blocks for the analysis.

Full Citation: C. Chen, L. Santinelli, J. Hugues, and G. Beltrame, "Static probabilistic timing analysis in presence of faults," in 2016 11th IEEE Symposium on Industrial Embedded Systems (SIES), May 2016, pp. 1–10.

3.2 Abstract

Accurate timing prediction for software execution is becoming a problem due to the increasing complexity of computer architecture, and the presence of mixed-criticality workloads. Probabilistic caches were proposed to set bounds to Worst Case Execution Time (WCET) estimates and help designers improve system resource usage. However, as technology scales down, system fault rates increase and timing behavior is affected. In this paper, we propose a Static Probabilistic Timing Analysis (SPTA) approach for caches with evict-on-miss random replacement policy using a state space modeling technique, with consideration of fault impacts on both timing analysis and task WCET. Different scenarios of transient and permanent faults are investigated. Results show that our proposed approach provides tight probabilistic WCET (pWCET) estimates and as fault rate increases, the timing behavior of the system can be affected significantly.

3.3 Introduction

A time-critical computing system, such as a satellite on-board computer, requires accurate timing prediction of software execution. If events are not managed within a certain time frame, the result may be catastrophic. A conservative estimation on execution time for traditional deterministic architecture will place the Worst Case Execution Time (WCET) far away from the actual maximum time used by the application [18].

To help predicting timing behavior, probabilistic real-time systems were introduced and such systems have very few pathological cases [90]. One method to realize probabilistic system is to modify the behavior of the cache – a bridge between processor and main memory – and make it random [90], which provides overall tighter bounds due to the lack of pathological cases. Two timing analysis techniques are proposed for systems with random caches: the Measurement Based Probabilistic Timing Analysis (MBPTA) and the Static Probabilistic Timing Analysis (SPTA). While MBPTA is based on repeated testing of an application for estimating its timing probability distribution, the SPTA uses detailed knowledge of software and hardware for obtaining a precise timing analysis with safe timing bounds.

As transistor size decreases, circuits become more sensitive to transient faults [36] which affect system timing behavior. Transient faults might be caused by high local temperatures or radiation effects, such as package alpha decay or galactic cosmic rays impacts, even at the ground level [85]. Furthermore, wear-out effects can introduce permanent faults throughout the lifetime of a device. Such effects are also exacerbated by technology scaling [50]. As a result, reliability to permanent and transient faults has to be considered for timing analysis.

In this paper, we present an SPTA methodology for instruction caches with random replacement policy, that takes both transient and permanent faults into consideration. Our methodology takes single-path program memory traces as inputs and computes probabilistic WCET (pWCET), i.e. exceedance probabilities with respect to execution time (the number of processor cycles in our simulations). The calculation is performed using state space techniques, and it is based on a non-homogeneous Markov chain model [97]. At every step, the current status of the system can be represented as a vector containing the probability of each state. The status of next step is computed using a transition matrix. To perform timing analysis, timing distribution vectors – which are used for timing representation and analysis – are assigned to each state. Transient and permanent fault effects are addressed as probabilistic models using fault injection. We employ an online fault detection mechanism for both faults and modify the system state at each step for accounting of faults. Our results show that by our approach, we can obtain tight pWCET estimates. In addition, different

scenarios of faults are studied. We can see that permanent faults have a significant impact on performance degradation.

The rest of the paper is organized as follows: related work is discussed in Section 7.4; Section 6.5 introduces system model based on Markov chain; the methodology using the model is explained in Section 5.6; fault models and their impacts on the system are demonstrated in Section 5.5; real-world benchmarks are evaluated in Section 6.7; and finally Section 7.10 draws some concluding remarks.

3.4 Related Work

Several works on SPTA have been proposed for caches with random replacement policy. Zhou [118] proposes a cache hit formula using reuse distance – the number of memory addresses accessed between two consecutive references to the same memory address – which simplifies computational complexity significantly. The probabilities for each cache access are made independent, and the final result is the convolution of all cache accesses. However, Cazorla *et al.* [27] and Altmeyer *et al.* [9] have found his methodology unsound. Quinones *et al.* [90] and Kosmidis *et al.* [66] give other formulae for random caches, while Cucu-Grosjean *et al.* [37] and Cazorla *et al.* [27] perform probabilistic timing analysis using these formulae. However, the formulae in [66] may overestimate the cache hit ratio [40].

Davis *et al.* [39] develop a formula using reuse distance only for evict-on-miss caches, and Altmeyer *et al.* [9] prove it to be optimal when only reuse distance is known. Multi-path programs are also analyzed by assuming that they are bounded. Besides, maximum preemption effects during program execution are taken into account for timing analysis. Altmeyer *et al.* [9] propose an exhaustive analysis approach. To reduce its computational complexity, this exhaustive approach can be combined with simplified formulae [11], resulting in an improved algorithm for SPTA. Griffin *et al.* [49] propose a methodology from the field of Lossy Compression and compare it with the method in [9]: by using *May* and *Must* Analysis, the result is more accurate with appropriate parameters. Lesage *et al.* [71] propose an SPTA for multi-math programs by using a conservative approach: cache states upper-bounds are calculated and paths are reduced according to worst-case execution path expansion. To demonstrate the impact of random caches, Abella *et al.* [4], Altmeyer *et al.* [11] and Lesage *et al.* [71] have done comparisons between caches using LRU and random replacement policy.

There are few studies on Probabilistic Timing Analysis (PTA) in the presence of faults. Slijepcevic *et al.* [98] study fault-tolerant systems, and combine it with PTA. *Degraded Test Mode* is proposed for random caches, which specifies requirements for hardware design and

test. By using *Degraded Test Mode*, real time systems can be analyzed with probabilities, and the pWCET is performed using MBPTA. Slijepcevic *et al.* extend the work in [99]. They propose an approach taking account of timing impacts of error detection, correction, diagnosis, and reconfiguration (DCDR) and degraded performance due to faults. They verify the timing behavior with different fault scenarios on critical real-time embedded systems and their work is based on MBPTA. Hardy and Puaut [56] present an SPTA-based methodology to calculate pWCET for instruction caches that contains only manufacturing permanent faults with LRU replacement policy. Permanent faults are detected by tests and cache blocks with permanent faults are disabled. A fault-free pWCET and miss probability distribution due to faults are initially computed separately. Then they are combined to form the pWCET with permanent faults. This method does not consider permanent faults that occur during program executions.

Our approach is the first method that calculates the timing behavior of caches with random replacement policy in presence of both transient and permanent faults. We consider permanent faults that happen during execution and are caused by device wear-out effects. This approach is based on SPTA and provides safe and tight pWCET estimates.

3.5 System Model

In this section, we present our SPTA model for random caches based on Markov chains. Our model is applied to a fully associative cache and it can be generalized to a set associative cache in which the analysis of each cache set can be performed separately as a fully associative cache. The model uses a memory trace as the input, and obtains a pWCET as the output. It is based on system states, which is similar to other accurate SPTA approaches [9, 49] for random caches. Different heuristics are applied for these approaches, and we apply an adaptive heuristic in the proposed approach.

A Markov chain is a mathematical framework that describes how a system moves from one state to another. If the future state of a system depends uniquely on the current state, such a system forms a Markov chain. The current state describes the status of the system, and the transition matrix explains how the system transits into the next state.

For a cache with evict-on-miss random replacement policy, every time a cache miss happens, a cache block is randomly selected and replaced with the new data (the term *data* is used to refer to the content of a memory address). As a result, there may be different data in the cache at different times, i.e. the memory layout of the cache changes with time. To describe the status of the system, s_i is defined as the memory layout of the cache and $|s_i|$ is

the number of elements in this state. Example 3.5.1 shows how to construct states from a trace of memory access by a task.

Example 3.5.1 *Suppose there is a task τ and a 2-way cache. The memory accesses from τ are a, b, c, a, b. Then we can define the state space as $s_0 = \emptyset$, $s_1 = \{a\}$, $s_2 = \{b\}$, $s_3 = \{c\}$, $s_4 = \{a, b\}$, $s_5 = \{a, c\}$, $s_6 = \{b, c\}$. We can see that all memory layouts are included in the states, and $|s_0| = 0, |s_i| = 1 : i = 1, 2, 3, |s_i| = 2 : i = 4, 5, 6$.*

Each program step can be seen as an access to a new memory address. Every time a memory address is accessed, the system advances by 1 time step and the system state may change. Each possible state of the system, i.e. memory layout, at a given step is associated with a probability.

The state occurrence probability vector $\overline{S}$ is defined as:

$$\overline{S} = [Pr(s_0), Pr(s_1), \cdots], \tag{3.1}$$

where $Pr(s_i)$ is the probability of the state s_i. With m being the number of different memory addresses in the program, and l being minimal value between cache associativity and m, the number of states can be calculated as:

$$\sum_{k=0}^{l} \begin{pmatrix} m \\ k \end{pmatrix}. \tag{3.2}$$

In addition to $\overline{S}$, we introduce the transition matrix $\overline{P}$, which describes how one state varies from the current step to the next. It is represented as:

$$\overline{P} = \begin{pmatrix} p_{0\to0}, & p_{0\to1}, & \cdots \\ p_{1\to0}, & p_{1\to1}, & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}, \tag{3.3}$$

where $p_{i\to j}$ is the probability for the system to go from state s_i to state s_j. In our model, $p_{i\to j}$ varies constantly, because it depends on the current system state and the memory accesses. At each step, the system may access different memory addresses and its state may change. Consequently, the transition probability $p_{i\to j}$ may change and this is a non-homogeneous Markov chain model.

Assuming $\overline{S}_k$ and $\overline{P}_k$ are the state probability vector and the transition matrix at step k,

respectively, then we have

$$\overline{S}_{k+1} = \overline{S}_k \overline{P}_k. \tag{3.4}$$

We can see that the state of the system for next step only depends on current state and the transition matrix.

3.6 Methodology

In this section, we demonstrate how to perform SPTA using the proposed system model based on Markov chains.

3.6.1 Transition Matrix Calculation

In our Markov chain model, Equation (3.4) is used to describe the system behavior. Given an initial state, we can calculate the transition matrix at each step and obtain the system state.

Algorithm 2 takes two inputs: the state occurrence probability vector $\overline{S}$ and the incoming memory address, and produces one output: the transition matrix $\overline{P}$. The algorithm checks all states and generates the transition matrix elements accordingly:

Line 4: All transition probabilities for state s_i are first initialized to 0. They may be modified later depending on current state and incoming address.

Line 6: $Pr(s_i) = 0$ means that state s_i is an impossible state at the current step. Therefore we have $\forall m, p_{i \to m} = 0$, i.e. one cannot exit from an impossible state.

Line 9: If s_i corresponds to an empty cache, a cache miss is inevitable, and there is only one possible transition from the empty cache state to the cache state with the incoming memory address.

Line 2: If the requested memory address is in the cache, there is a cache hit. In this case, the cache will not change its state with probability 1, i.e. $p_{i \to i} = 1$.

Line 5: If the requested memory address is not in the cache, there is a cache miss and the transition matrix is computed. This is the most complex case: the new memory address may replace an existing cache block, or it may be put into a new cache block and probabilities have to be computed accordingly. In our target cache, the probability of replacing an existing cache block is $1/N$ (see Line 8), where N is the cache associativity. This is because we consider an evict-on-miss random cache, and a cache block is

ALGORITHM 1: Transition matrix calculation

Data: State prob. vector $\overline{S}$, memory address a
Result: Transition matrix $\overline{P}$

$n \leftarrow |\overline{S}|$; //number of states in $\overline{S}$;
for $i \leftarrow 0$ **to** *n-1* **do**
  for $j \leftarrow 0$ **to** *n-1* **do**
    $p_{i \to j} \leftarrow 0$; //*initialize transition matrix*
  end
  if $Pr(s_i) = 0$ **then**
    go to next i; //*state* s_i *does not exist*
  end
  if $s_i = \emptyset$ **then**
    $p_{i \to m} \leftarrow 1$; //$s_m = \{a\}$, *cache miss for* s_i
    go to next i;
  end
  if $a \in s_i$ **then**
    $p_{i \to i} \leftarrow 1$; //*cache hit*
    go to next i;
  end
  $ind0 \leftarrow \emptyset$; //*indexes of transitions by replacement*
  $ind1 \leftarrow \emptyset$; //*indexes of transitions by new cache block*
  $q \leftarrow |s_i|$; //*number of addresses for state* s_i
  for $j \leftarrow 0$ **to** *n-1* **do**
    $p \leftarrow |s_j|$; //*number of addresses for state* s_j
    if *p-q=0* **then**
      $l \leftarrow |s_i - s_j|$; //*number of different addresses*
      if *l=1* ***and*** $a \in s_j$ **then**
        Add j to $ind0$; //*replaces existing address*
      end
    end
    if *p-q=1* **then**
      if $s_i \subset s_j$ ***and*** $a \in s_j$ **then**
        Add j to $ind1$; //*add new address*
      end
    end
  end
  $N \leftarrow$ cache associativity;
  for $x \in ind0$ **do**
    $p_{i \to x} = 1/N$; //*replacement prob.*
  end
  for $x \in ind1$ **do**
    $p_{i \to x} = (N - q)/N$; //*new address prob.*
  end
end

randomly selected for replacement with probability $1/N$. The probability for a memory address to be placed in an empty cache block is $(N-q)/N$ (see Line 12), where q is the number of blocks in use for the current state s_i. This is due to the fact that if the new memory address does not cause a replacement, it can only be put into an empty cache block. The number of empty cache blocks is $N-q$, and they are chosen from N ways. Therefore the probability is $(N-q)/N$.

Example 3.6.1 shows how to obtain the state at a given step for Example 3.5.1 using the transition matrix using Equation (3.4) and Algorithm 2 .

Example 3.6.1 *Let $p_{i\to j} = 0$ at the beginning of each step, and assuming the cache is initially empty at the beginning of step 1, then we have $\overline{S}_1 = [1,0,0,0,0,0,0]$.*

At step 1: For $\overline{P}_1$, all elements are 0 except $p_{0\to 1} = 1$. $\overline{S}_2 = \overline{S}_1 \cdot \overline{P}_1 = [0,1,0,0,0,0,0]$.
At step 2: $p_{1\to 2} = 1/2, p_{1\to 4} = 1/2$, $\overline{S}_3 = \overline{S}_2 \cdot \overline{P}_2 = [0,0,1/2,0,1/2,0,0]$.
At step 3: $p_{2\to 3} = 1/2, p_{2\to 6} = 1/2, p_{4\to 5} = 1/2, p_{4\to 6} = 1/2$,
$\overline{S}_4 = \overline{S}_3 \cdot \overline{P}_3 = [0,0,0,1/4,0,1/4,1/2]$.
At step 4: $p_{3\to 1} = 1/2, p_{3\to 5} = 1/2, p_{5\to 5} = 1, p_{6\to 4} = 1/2, p_{6\to 5} = 1/2$, $\overline{S}_5 = \overline{S}_4 \cdot \overline{P}_4 =$ $[0,1/8,0,0,1/4,5/8,0]$.
At step 5: $p_{1\to 2} = 1/2, p_{1\to 4} = 1/2, p_{4\to 4} = 1, p_{5\to 4} = 1/2, p_{5\to 6} = 1/2$, $\overline{S}_6 = \overline{S}_5 \cdot \overline{P}_5 =$ $[0,0,1/16,0,5/8,0,5/16]$.

3.6.2 Timing Analysis

With Algorithm 2, Equation (3.4) can be used to describe the system state transitions. Nonetheless, the duration of a task execution is different from the step used in the Markov chain. At each step, one memory address is accessed and different number of cycles may be applied to the timing analysis according to the system state. Without loss of generality, we assume 1 cycle for a cache hit and 100 cycles for a cache miss (any timing behavior would work). With different number of cycles executing the program and their corresponding occurrence probabilities, we have timing distributions for programs. The resulting timing distributions are discrete-time distributions as each memory access takes $n \in \mathbb{N}$ number of cycles.

Two vectors are introduced for defining timing distributions and modeling timing behaviors with respect to probabilities. A cycle vector $\overline{C}$ can be used to denote the timing distribution in terms of number of cycles, and a probability vector $\overline{M}$ can represent the probability of

occurrence for $\overline{C}$. Then we have $\overline{C} = [c_0, c_1, \cdots]$ and $\overline{M} = [m_0, m_1, \cdots]$, where $c_i \in \mathbb{N}$ represents the program duration in cycles and $m_i = Pr(c_i), m_i \in \mathbb{R}$ denotes the occurrence probability for c_i. A scalar addition of $\overline{C}$ and a scalar multiplication of $\overline{M}$ are defined as
$\overline{C} + n = \{c + n | n \in \mathbb{N},\, c \in \overline{C}\}$,
$\overline{M} \cdot p = \{m \times p | p \in \mathbb{R}, m \in \overline{M}\}$.

With the cycle vector $\overline{C}$ and its probability vector $\overline{M}$, we define the timing distribution for state s_i as $\mathcal{T}_i =< \overline{C}_i, \overline{M}_i >$. $\mathcal{T}_i$ collects the number of cycles to execute the program and corresponding probabilities for each cycle. The timing distribution changes during program execution; the initial values is $\mathcal{T}_i =< [0], [1] >$, and each memory access adds additional cycles and probabilities to the timing distribution.

Timing distributions $\mathcal{T}$ need to be combined during state transitions, since different states can transit to the same state after the memory access. Therefore the merge operation between timing distributions $\uplus$ is defined such that the resulting distribution $\mathcal{T}_k$ is

$$\mathcal{T}_k = \mathcal{T}_i \biguplus \mathcal{T}_j =< \overline{C}_k, \overline{M}_k >, \tag{3.5}$$

where $\overline{C}_k = \overline{C}_i \cup \overline{C}_j$, and $\overline{M}_k = \{m_p + m_q | m_p \in \overline{M}_i, m_q \in \overline{M}_j, c_p \in \overline{C}_i, c_q \in \overline{C}_j, c_p = c_q\}$. The merge operation puts all number of cycles into one vector, and the probabilities with the same number of cycle are added together.

With the transition matrix $\overline{P}$, the timing distribution for state s_j is:

$$\mathcal{T}_j = \begin{cases} < \emptyset, \emptyset > & if \quad \forall i, p_{i \to j} = 0 \\ \biguplus_i < \overline{C}_i + n_a, \overline{M}_i \cdot p_{i \to j} > & otherwise, \end{cases} \tag{3.6}$$

where

$$n_a = \begin{cases} n_h & if \quad j = i \\ n_m & if \quad j \neq i, \end{cases} \tag{3.7}$$

n_h is the number of cycles for a cache hit, and n_m is the number of cycles for a cache miss, e.g. the previously chosen values of 1 and the 100.

By merging timing distribution vectors of all states, using Equation (3.6), we could compute the timing distribution of the whole program as $\mathcal{T} =< \overline{C}, \overline{M} >$.

Having the timing distribution, for each state and for the whole program, we can compute both the Cumulative Distribution Function and the inverse Cumulative Distribution Function (1-CDF) [39]. The inverse cumulative is an exceedance function showing the probability of exceeding a certain program duration in cycles. The 1-CDF probabilities are denoted as

$\overline{Q} = [q_0, q_1, \cdots]$, with

$$q_i = \sum_{m_j \in \overline{M}, c_i, c_j \in \overline{C}, c_j > c_i} m_j. \tag{3.8}$$

The inverse timing distribution is defined as $\mathcal{I} =< \overline{C}, \overline{Q} >$ and can be related to the state s_i, i.e. $\mathcal{I}_i$, or the whole program, i.e. $\mathcal{I}$.

Example 3.6.2 *In this example we demonstrate how to do timing analysis for the parameters listed in Example 3.5.1.*

At each step, we use Equation (3.6) *to compute* $\mathcal{T}_i$. *Let* $n_h = 1$, $n_m = 100$

At step 1: $\mathcal{T}_1 =< [100], [1] >$.
At step 2: $\mathcal{T}_2 =< [200], [1/2] >$, $\mathcal{T}_4 =< [200], [1/2] >$.
At step 3: $\mathcal{T}_3 =< [300], [1/4] >$, $\mathcal{T}_5 =< [300], [1/4] >$, $\mathcal{T}_6 =< [300], [1/2]$.
At step 4: $\mathcal{T}_1 =< [400], [1/8] >$, $\mathcal{T}_4 =< [400], [1/4]$, $\mathcal{T}_5 =< [301, 400], [1/4, 3/8] >$.
At step 5: $\mathcal{T}_2 =< [500], [1/16] >$, $\mathcal{T}_4 =< [401, 500], [3/8, 1/4]$, $\mathcal{T}_5 =< [401, 500], [1/8, 3/16] >$.

By Equation (3.6), *we have* $\mathcal{T} =< [401, 500], [1/2, 1/2] >$.

From Equation (3.8), *the inverse timing distribution is* $\mathcal{I} =< [401, 500], [1/2, 0] >$, *i.e. there is the probability of* $1/2$ *to exceed 401 cycles, and the probability to exceed 500 cycles is* 0, *since the maximum execution time is 500 cycles.*

It is worth noting that for a set associative cache, the Markov chain model applies to each cache set CS_k. As a result, there are both the timing distribution $\mathcal{T}_{CS_k}$ and the inverse timing distribution $\mathcal{I}_{CS_k}$ specific of the cache set. Assuming a deterministic placement policy (e.g. modulo placement) is applied, let a_m^k be an address assigned to cache set CS_k. This address can assigned to only one cache set. The timing distribution $\mathcal{T}_{CS_k}$ is a function of the addresses assigned to it, i.e. $\mathcal{T}_{CS_k} = f(a_0^k, a_1^k, ...)$. For another cache set timing distribution, we have $\mathcal{T}_{CS_l} = f(a_0^l, a_1^l, ...)$ and $a_m^k \neq a_n^l : k \neq l$. We can see that cache set timing distributions are functions of different addresses and are thus statistically independent of each other, i.e. $\mathcal{T}_{CS_k} \perp \mathcal{T}_{CS_l} : k \neq l$.

To obtain the timing distribution $\mathcal{T}$ of different cache sets, we apply the convolution operator $\otimes$ between different cache set timing distributions:

$\mathcal{T}_{CS} = \mathcal{T}_{CS_k} \otimes \mathcal{T}_{CS_l} =< \overline{C}, \overline{M} >$,

where $\overline{C} = \{c_p + c_q | c_p \in C_{CS_k}, c_q \in C_{CS_l}\}$, and $m_i \in \overline{M}$ calculated as $m_i = \sum_{\substack{m_p \in \overline{M}_{CS_k} \\ m_q \in \overline{M}_{CS_l} \\ c_p + c_q = c_i}} m_p m_q$.

3.6.3 Adaptive Method

The result of our Markov model is an accurate timing analysis, because it takes all states into account and computes how they change over time. The Markov model overcomes the pessimism introduced by formulae in [39]. The resulting timing distribution $\mathcal{T}$ is the pWCET obtained accounting for all the cache configurations while the program executes.

However, from Equation (3.2) we can see that the number of states increases polynomially with a high exponent value as more memory addresses are accessed. The method proposed could become intractable. We use then an adaptive method to limit the number of states and to produce a result with reasonable accuracy, which scales with the size of memory accesses.

Suppose there are n different memory addresses, in order to reduce computational complexity, we would like to use only m ($m < n$) memory addresses for the states so that the number of states is limited and we can enumerate all states. This is realized with two parts: the i) *state modification* and the ii) *state and timing distribution merge*.

i) *State modification*: for the first m different addresses $a_0, a_1, ...a_{m-1}$, where $a_i \neq a_j$ for $i \neq j$. We construct the state space $\{s_0, s_1, ...\}$ using the proposed Markov chain method. We have $\forall A \subseteq \{a_0, a_1, ...a_{m-1}\}$, $\exists i : A \subseteq s_i$. The number of states is from Equation (3.2). When another new memory address a_m comes, we modify states in the state space, instead of increasing the number of states.

In order to modify states, we find a memory address $a \in \{a_0, a_1, ...a_{m-1}\}$. The state s_i containing a is changed to state s_j in which a_m replaces a. There are different heuristics to select a. We assume that a least recently used address a has a low probability to be used again in recent memory accesses, and a is to be replaced as follows: $\forall i : a \in s_i, s_j = s_i \setminus \{a\} \cup \{a_m\}$. The probability and the timing distribution for s_j are respectively $Pr(s_j) = 0$ and $\mathcal{T}_j =< \emptyset, \emptyset >$. This way, the number of states remains the same, but different addresses can be used in the state space. The method works in an adaptive way by modifying the states with the least recently used addresses. Note that the least recently used address is used to change states, and it is not a cache replacement policy.

ii) *State and timing distribution fusion*: When states are changed, we need to take timing analysis into account as well, because each state is assigned different timing distributions. To obtain the safe bound to the pWCET, we use a conservative method dealing with the $\overline{S}$ and $\mathcal{T}$ variables for the timing analysis. $\overline{S}$ is the state occurrence vector and $\mathcal{T}$ is the timing distribution.

Suppose $s_i : a \in s_i$ is the state before state modification, and s_p is the state containing all memory addresses in s_i except the address a that is to be replaced, i.e. $s_p = s_i \setminus \{a\}$. In

state modification, we have seen that whenever a new address is accessed, we may change the state s_i. Therefore the state vector which represents its occurrence probability must be modified accordingly. In the new state vector $\overline{S}$, we use

$$Pr(s_p) = Pr(s_i) + Pr(s_p) \tag{3.9}$$

The occurrence probability $Pr(s_p)$ can be accumulated to account for the occurrence probability $Pr(s_i)$, because if $s_p \subseteq s_i$, for any new memory address, state s_i has the same or a higher cache hit probability compared to that from state s_p. Equation (3.9) adds pessimism to timing analysis, but it provides a safety bound. In addition to state modification, we need to merge the timing distribution $\mathcal{T}_i$ to $\mathcal{T}_p$ using Equation (3.5).

After the *state modification* and *state and timing distribution fusion*, the Markov chain model uses the same methodology developed in Section 5.6 for new memory accesses. By using the state space constructed by m addresses, the adaptive method is tractable. To trade off for tractability, the accuracy is compromised, because only some of addresses are used to build the state space. Therefore timing behaviors from the addresses that are discarded are not considered. Nevertheless, the pWCET estimate from this method is safe and it becomes tighter as more addresses are applied.

3.7 Fault Impacts

In this section, both transient and permanent fault models are introduced to the system that is equipped with an online fault detection mechanism. We consider faults that only occur in the storage elements of the cache and apply probabilistic models for faults. Faults in combinational circuits are not considered in this paper. We use the fault occurrence probability of each memory access step for analysis. Since a cache miss takes longer than a hit, to simplify the analysis and obtain a safe bound, we assume that each memory access step is a cache miss, i.e. it takes n_m cycles and this value is used for fault rate calculation in following sections.

For set associative caches, different cache sets may be accessed. Let n_i and n_{i+1} be consecutive steps to access the same cache set, and n_s be the step difference. We have

$$n_s = n_{i+1} - n_i. \tag{3.10}$$

For fully associative caches, $n_s = 1$. We assume a constant fault rate f is applied to both

transient and permanent faults. The probability to have a fault for one cache block is

$$1 - (1 - f)^{n_s}, \tag{3.11}$$

and the probability without a fault is $(1 - f)^{n_s}$.

3.7.1 Transient Fault Impact

A Single Event Upset (SEU) is a change of state caused by a high-energy particle. We regard an SEU as a transient fault, since it does not cause permanent damage and the system can be recovered. SEUs are known to be independent, and we assume they are uniformly distributed in space and time, i.e. each cache block has the same fault rate throughout the program execution. After the fault occurs, the cache block remains faulty unless this fault is detected. This is because the transient fault happens to the storage element. Once the storage state changes, it can not recover automatically. As a result, transient fault effect lasts. After the fault is detected, this block is seen as invalid, but it does not affect following data storage in it.

To deal with transient fault, many techniques have been proposed. For example, Reed-Solomon codes have been used extensively for space applications to detect and correct transient fault errors. In this paper, we employ a simple parity check fault detection mechanism – where parity bits are are added to the data – to first level (L1) cache, which has less area and speed penalties for L1 caches compared to commonly used single error correction-double error detection (SEC-DED) techniques [79].

When a cache block is accessed, the parity bits stored are examined to see if any transient fault has occurred. Due to low probability of fault, we assume that all faults can be detected. If any fault is detected, the corresponding data is regarded as invalid and will be fetched from the main memory again.

We note that with parity check mechanism, the impact of a transient fault is equivalent to a cache eviction, i.e. once the transient fault occurs, the data is not valid any more and it is a cache miss. Let f_t be the transient fault probability at each step and s_i be the state before transient fault detection. After fault detection, s_i becomes s_p. With Equation (3.10) and (4.4), we have $s_p \subseteq s_i$ and

$$Pr(s_p) = Pr(s_i)((1 - f_t)^{n_s})^{|s_p|}(1 - (1 - f_t)^{n_s})^{|s_i|-|s_p|}. \tag{3.12}$$

Transient fault detection produces new states, which may be the same as existing ones. If

$\exists s_m : s_m = s_p$, we change timing distributions by multiplying the state change probability, and then merge state probabilities and timing distributions using Equation (3.9) and Equation (3.5). Example 3.7.1 demonstrates how a state changes because of transient fault after one step.

Example 3.7.1 *Suppose we have a state* $s = \{a, b\}$, $Pr(s) = 0.5$ *and the transient fault probability is* $f_t = 0.1$ *at each step. After one step, from Equation* (3.12) *we know that new states are produced and we have*

$$\begin{aligned} Pr(s = \{a, b\}) &= 0.5 \times (1 - 0.1)^2 = 0.405 \\ Pr(s = \{a\}) &= 0.5 \times 0.1 \times (1 - 0.1) = 0.045 \\ Pr(s = \{b\}) &= 0.5 \times 0.1 \times (1 - 0.1) = 0.045 \\ Pr(s = \emptyset) &= 0.5 \times 0.1^2 = 0.005 \end{aligned}$$

3.7.2 Permanent Fault Impact

Permanent Fault Model

Permanent faults are faults whose effects are assumed to last from the moment they appear to the end of the program execution. When a permanent fault occurs to a cache block, it cannot be used any more.

To model permanent faults we start by defining the probability $f_p(t, T)$ of a permanent fault occurring. It is the probability of fault in a system component by time t, $failure \leq t$, given that the component was still functional at the end of the previous interval $t - T$, $failure > t - T$. T is the scrubbing period, i.e. the time interval between two consecutive fault detection to avoid error accumulation. In this paper it is the time for one memory access. This probability can be computed using the Kolmogorov definition from the formula in [86], as:

$$\begin{aligned} f_p(t, T) &= Pr(failure \leq t | failure > t - T) \\ &= \frac{Pr(failure \leq t \wedge failure > t - T)}{Pr(failure > t - T)} \\ &= \frac{\mathsf{cdf}_{failure}(t) - \mathsf{cdf}_{failure}(t - T)}{1 - \mathsf{cdf}_{failure}(t - T)}, \end{aligned} \tag{3.13}$$

with $\mathsf{cdf}_{failure}$ the cumulative density function of the random variable failure describing the time at which the failure happens.

In literature, several probability distributions are used to model failure times [86]. One of the most frequently used is the exponential distribution; however, the exponential distribution representation is somewhat imprecise because it lacks the ability to capture the increasing failure probability due to accumulated wear in the component. A common alternative used to overcome this limitation is a log-normal failure distribution:

$$f_p(t,T) = \frac{\mathsf{cdf}_{norm}\frac{ln(t)-\mu}{\sigma} - \mathsf{cdf}_{norm}(\frac{ln(t-T)-\mu}{\sigma})}{1-\mathsf{cdf}_{norm}(\frac{ln(t-T)-\mu}{\sigma})}, \tag{3.14}$$

where cdf_{norm} the cumulative density function of the normal distribution. The mean and standard deviation parameters of such distribution can be computed from the Mean Time To Failure (MTTF) such that

$$\begin{aligned} \mu &= ln(\frac{MTTF^2}{\sqrt{var_{MTTF}+MTTF^2}}) \\ \sigma &= \sqrt{ln(1+\frac{var_{MTTF}}{MTTF^2})}. \end{aligned} \tag{3.15}$$

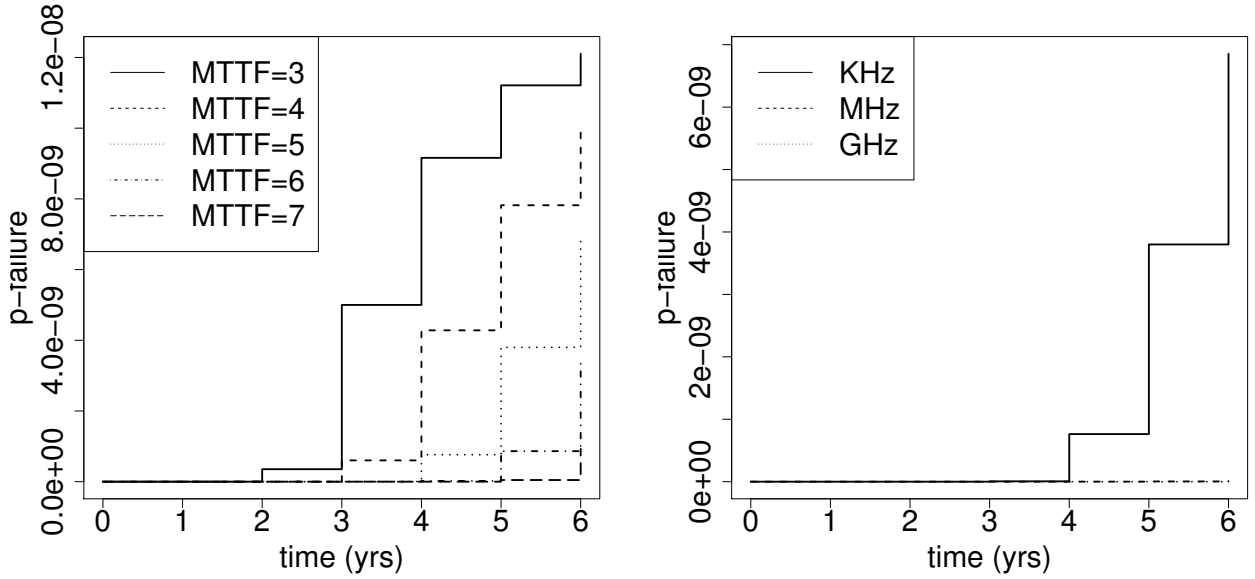
Note that in Equation 3.14, $f_p(t,T)$ depends on the actual time t and the scrubbing period T. The non-memoryless distribution function describe the occurrence of a recent failure with larger probability than a memory-less distribution like the exponential one.

Figure 3.1(a) summarizes the comparison of the log-normally distributed failure times with different MTTFs. The plot is discretized in years. We can see that as MTTF increases, the permanent fault rate f_p for each memory access decreases, because a smaller fault rate can lead to a longer lifetime. In addition, f_p is an increasing function of time. As system operation time increases, f_p increases continuously. In this paper, however, we assume that f_p is constant, because execution times of our benchmarks are short and f_p rises extremely slowly.

Figure 3.1(b) depicts the comparison of the log-normally distributed failure times with different operating frequencies. The desired MTTF is arbitrarily set at 5 years. At the beginning, the fault rate is extremely low, e.g. with KHz frequency, the permanent rate is 4.4×10^{-18} at year 2. The MHz and GHz fault rates are 10^{-3} and 10^{-6} smaller respectively.

Permanent Fault Detection

To deal with permanent fault in the SPTA, we establish different Markov chain models with different numbers of faults. For N-way set associative caches, we implement $N+1$ Markov chain models, where the ith model contains $i-1$ permanent faults and there are $N-(i-1)$



(a) Varying the MTTF. Increasing the MTTF the f_p in years (yrs) decreases consistently. f_p computed from a KHz frequency

(b) Varying the platform frequency. The f_p in years (yrs) for MHz and GHz are respectively 10^{-3} and 10^{-6} smaller than those for KHz

Figure 3.1 Failure probability f_p the MTTF and the platform frequency

available cache blocks. The model with N faults is the worst case where all cache blocks are faulty. Its timing analysis is easy to calculate since there are always cache misses.

When the memory is accessed, fault detection is applied using parity check. To identify if the fault is permanent, we adopt the method proposed by [2]. It is simple to implement and tracks the number of fault occurrences. If a threshold value is exceeded, the fault is classified as permanent. To improve following timing behaviors, we assume that each cache block can be controlled separately. Once a cache block is classified to have a permanent fault, it will be disabled and will not be used any more.

Figure 3.2 shows how to apply different Markov chain models to N-way caches, with each node representing a Markov chain model state space. There are $N + 1$ rows for N-way caches, i.e. $N + 1$ Markov chain models. Memory addresses are accessed at step $1, 2, 3, ...$ The analysis is completed in two phases to account for fault events.

Phase 1: Fault detection. This is denoted by dotted lines. Node $\overline{S}_n^m$ denotes the state occurrence probability vector of the system with m faults at step n. Let f_p be the probability of permanent fault at each step. With Equation (3.10) and (4.4), for each node, the state $s_i^m \in \overline{S}_n^m$ is changed as follows.

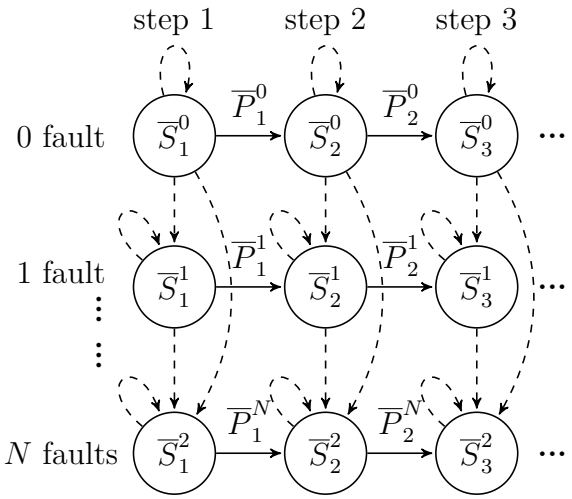


Figure 3.2 Different Markov chain models taking account of permanent faults for N-way caches. Dotted lines indicate state changes due to fault detection, and solid lines denote state transitions due to memory accesses.

- No permanent faults occur.

$$Pr(s_i^m) = Pr(s_i^m)((1 - f_p)^{n_s})^{N-m}. \tag{3.16}$$

  The probability of this state changes due to potential permanent fault occurrences, and timing distribution are changed by multiplying the state change probability.

- Permanent faults occur. Let l be the number of added permanent faults on current model and $s_p^{m+l} \in \overline{S}_n^{m+l}$ be the state after permanent fault detection. We assume that permanent faults can be detected immediately after they appear. Then we have $s_p^{m+l} \subseteq s_i^m$ and

$$Pr(s_p^{m+l}) = Pr(s_i^m)((1 - f_p)^{n_s})^{N-m-l}(1 - (1 - f_p)^{n_s})^l. \tag{3.17}$$

  For state $s_m^{m+l} : s_m^{m+l} = s_p^{m+l}$, we change timing distributions by multiplying the state change probability, and then merge state probabilities and timing distributions using Equation (3.9) and Equation (3.5).

Phase 2: State transition. This is denoted by solid lines. After the fault detection, states in different Markov chain models are updated, since new addresses are accessed. Together with the transition matrix $\overline{P}_n^m$ (the transition matrix with m faults at step n), the timing analysis methodology from Section 5.6 is applied to each model.

3.8 Experimental Results

3.8.1 Experimental Setup

In our experiments, we used the SoCLib open platform[1] to generate memory traces for benchmarks. The platform is equipped with one MIPS 32-bit processor with L1 instruction cache. In order to evaluate our approach, we adopt Mälardalen benchmarks [52], a popular benchmark suite used for WCET evaluation and analysis. Due to limited space, we only present results of two benchmarks ($fdct$ and crc). By injecting faults into instruction cache with different fault rates, we investigate their impacts on the system. The cache size is 512 bytes, with 2-way associativity and 4-byte cache block. We assume that for each cache miss, the duration is 100 cycles; for each cache hit, the duration is 1 cycle. To account for fault detection delays, we add additional 10 cycles. A cache with bigger size can also be adopted, in which more cache sets are used and thus there are fewer addresses for each set. The timing distribution calculations for each set perform faster due to address reductions.

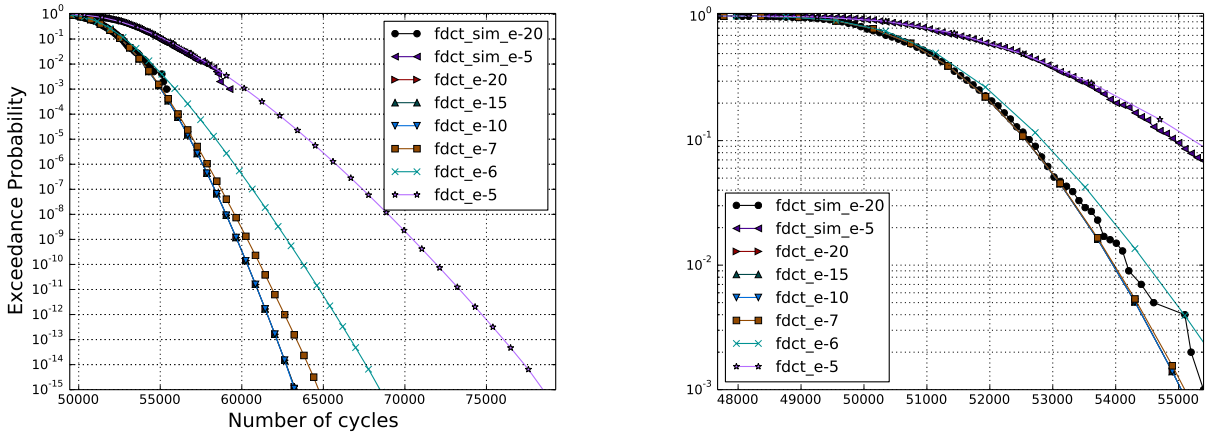
In our adaptive Markov chain model, we adopt 4 memory addresses for adaptive state modification. Since execution times of benchmarks are short, we create synthetic benchmarks which repeat the same benchmark 10 times. This way, we can study as execution time increases, how the system is affected by the faults and we can see if repeated benchmarks produce similar pWCET estimates to the original benchmarks. We perform 1,000 simulations for each benchmark as the base line and this can be used to verify the accuracy of the method at around exceedance probability of 10^{-3}. A brief comparison can be done using such an exceedance probability between simulations and our approach. If a lower exceedance probability is required, more simulations can be performed. [2]

Figure 3.3 – Figure 3.6 show timing analyses of benchmarks $fdct$ and crc respectively. $fdct$ is fast discrete cosine transform using a lot of calculations based on integer arrays and crc is cyclic redundancy check computation using complex loops with lots of decision. On each figure, the x-axis shows the number of cycles and y-axis represents the exceedance probability (i.e. 1-CDF) for corresponding cycles. The exceedance probability is set as 10^{-15}, for the failure rate requirement at the highest level for commercial airborne is translated into the region of around 10^{-13}. To show simulation results in detail, a zoomed figure of each benchmark which limits the exceedance probability to 10^{-3} is displayed. Different fault scenarios are applied. The transient fault rate that we applied is at each step, there is a probability of 10^{-20} for fault occurrence. Different permanent fault rates are applied to

[1] http://www.soclib.fr

[2] The replication package of our method script is available on demand.

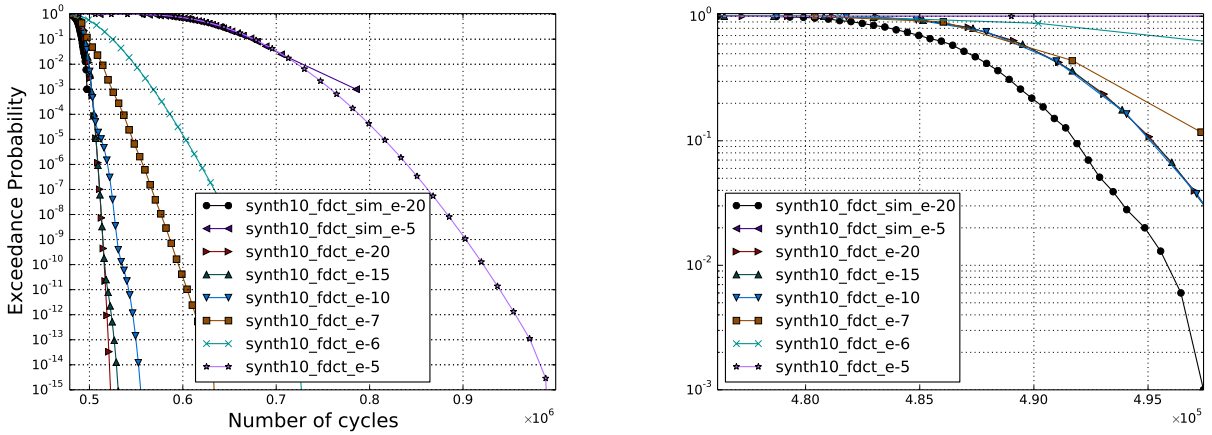
show account for wear-out effects at different times. Since for a MHz with 5-year MTTF, the permanent fault probability is around 10^{-20}, we applied permanent fault probabilities of $10^{-20}, 10^{-15}, 10^{-10}, 10^{-7}, 10^{-6}, 10^{-5}$ respectively to study how the system is affected when permanent fault rate increases.



(a) original fdct

(b) zoomed original fdct

Figure 3.3 fdct and the zoomed figure



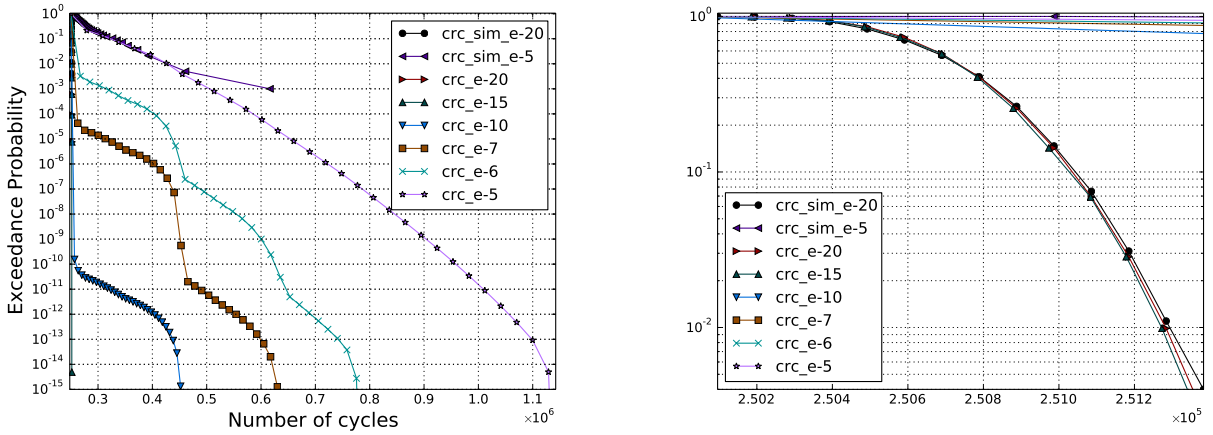
(a) synthetic (repeated) fdct

(b) zoomed synthetic (repeated) fdct

Figure 3.4 synthetic fdct and the zoomed figure

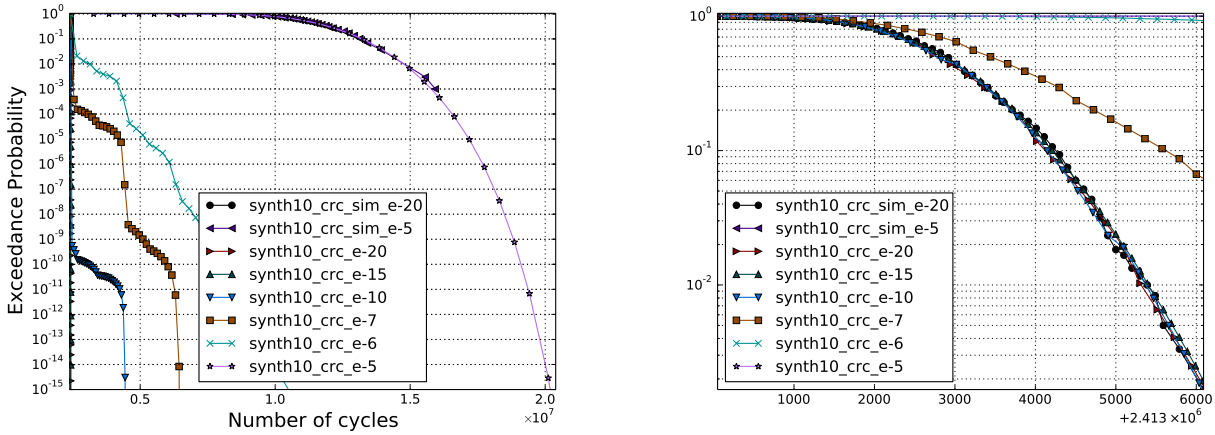
3.8.2 Discussion

To verify the accuracy of our SPTA approach, simulations are performed with transient fault rate at 10^{-20} per memory access, and two permanent fault rates at 10^{-20} and 10^{-5}



(a) original crc (b) zoomed crc

Figure 3.5 crc and the zoomed figure



(a) synthetic (repeated) crc (b) zoomed synthetic (repeated) crc

Figure 3.6 synthetic crc and the zoomed figure

per memory access are applied. From zoomed figures (Figure 3.3(b), 3.5(b), 3.6(b)), we can see that for all fault scenarios, at any exceedance probability the simulation execution time matches our result, which means that our approach provides tight pWCET estimates. Note that simulations results may show an error at low probabilities as the number of datapoints are insufficient to accurately estimate the exceedance function.

In Figure 3.4, we can see that there is a slight difference between simulations and the result from our approach, because our adaptive method uses only some of states for analysis. As a consequence, the accuracy may be compromised. Although the synthetic benchmark is a

repetition of an original benchmark, its timing behavior may be different. At an exceedance probability, the number of cycles for the synthetic benchmark is not the original value multiplied by the number of repetitions, because after the first completion of the benchmark, some code may exist in the cache, which can help reduce execution time for following executions.

For transient fault, its fault rate is extremely low. In addition, when the cache block with transient fault is accessed, the transient fault will be detected if it has occurred. The new data to be put into this cache block will not be affected, since transient fault does not accumulate. As a result, the impact of transient fault is not significant.

The fact that permanent faults can accumulate has a significant impact on the behavior of the system, especially since device aging can increase their rate. In our experiments, we have applied different permanent fault rates to the system. We can see that when the permanent fault rate is extremely low, the system is not affected during benchmark execution. However, as fault rate increases, the system takes more time to finish the benchmark.

The size of benchmarks has different impacts on transient and permanent faults. For transient faults, since they can be recovered, the benchmark size does not have a big influence. However, since permanent faults accumulate, as benchmark size increases, the execution times may become longer. For example, at the exceedance probability of 10^{-15}, for original $fdct$ with permanent fault rate of 10^{-5}, it takes around 80,000 cycles. Its synthetic benchmark takes around 1000,000 cycles. Even if it may contain some existing code for repetitions, the synthetic benchmark takes more than 10 times in terms of cycles due to permanent fault effects.

We note that for some permanent fault rates, there may be a sudden drop in exceedance probability, especially in Figure 3.5, where the exceedance decrease drops at similar execution time. This is because permanent faults may have significant impacts on some cache sets depending on benchmark characteristics. In Figure 3.7, two cache set exceedance probabilities are convolved, where x-axis is execution time and y-axis indicates exceedance probability. It shows that one cache set exceedance probability changes gradually while the other one is affected badly by permanent faults. As a result, the convolution result may have drastic drop around some execution time.

From the experiments, we can see that our approach can work with different fault rates. Depending on characteristics of benchmarks, their pWCET estimates are affected in different ways. For example, the $fdct$ benchmark pWCET varies gradually as higher-level permanent fault rates are applied, while *crc* benchmark exhibits pWCET by a dramatic change at some exceedance probabilities. One potential use of our approach is to estimate fault impacts on program timing behaviors with random replacement caches, so that we can make sure that

safety requirement is met under different fault scenarios.

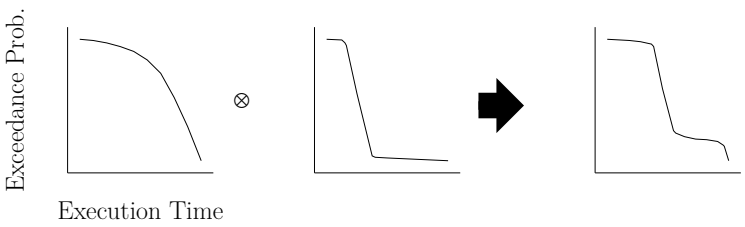


Figure 3.7 Convolution of two different exceedance probabilities. One exceedance probability decreases gradually, and the other decreases dramatically due to fault impacts.

3.9 Conclusion

In this paper, we have demonstrated an adaptive Markov chain based Static Probabilistic Timing Analysis (SPTA) methodology in presence of faults; our methodology is based on a non-homogeneous Markov chain model. Both transient and permanent faults are then introduced into the system. The states are modified accordingly for including fault impacts and the pWCET obtained embeds faults effects. The experimental results show how faults affect execution time.

In order to reduce computational complexity, the state space can be limited to the specified level. The state space is modified in an adaptive way, such that existing addresses can be replaced by new incoming addresses in the state space. This guarantees good accuracy and scalability of our SPTA analysis.

As future work, we intend to address aspects such as benchmark and platform fault tolerance. Furthermore, we will enhance our SPTA Markov chain methodology to apply preemptions and multi-processor embedded system platforms.

CHAPTER 4 ARTICLE 2: EFFECTS OF ONLINE FAULT DETECTION MECHANISMS ON PROBABILISTIC TIMING ANALYSIS

4.1 Preface

In our previous research [31], we have studied the transient and permanent faults that may occur in random caches. To preserve performance, it is necessary to detect transient faults and correct them. On the other hand, when a permanent fault occurs in a cache block, the block cannot be used to store data any more. Hence, we must detect such blocks and disable them to avoid recurring errors. A traditional rule-based online fault detection technique periodically counts the number of faults to classify a fault as transient or permanent. In this article, we introduce a permanent fault detection technique using Dynamic Hidden Markov Model (D-HMM) that predicts whether a cache block is permanently faulty or not using the detected faults and a "belief" threshold. The results of our experiments indicate that the D-HMM based detection technique dramatically improves system performance compared to the traditional rule-based detection technique.

Full Citation: C. Chen, J. Panerati, and G. Beltrame, "Effects of online fault detection mechanisms on probabilistic timing analysis," in 2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Sept 2016, pp. 41–46.

4.2 Abstract

In real time systems, random caches have been proposed as a way to simplify software timing analysis, by avoiding corner cases usually found in deterministic systems. Using this random approach, one can obtain an application's probabilistic Worst Case Execution Time (pWCET) to be used for timing analysis. As with deterministic systems, technology scaling in cache memories is making transient and permanent faults more likely, which in turn affects the system's timing behavior. To mitigate these effects, one can introduce a detection mechanism that classifies a fault as transient or permanent, with the goal of disabling permanently faulty cache blocks to avoid future accesses. In this paper, we compare the effects of two online detection mechanisms for permanent faults, namely rule-based detection and Dynamic Hidden Markov Model (D-HMM) based detection, for the generation of safe pWCET estimates. Experimental results show that different mechanisms can greatly affect safe pWCET margins, and that by using D-HMM the pWCET of the system can be improved compared

to rule-based detection.

4.3 Introduction

In safety-critical systems (e.g., the on-board computers of airplanes and spacecrafts), timing behaviors must be analyzed to ensure that the systems meet the safety requirements. Numerous methods have been proposed to compute the Worst Case Execution Time (WCET). However, as technology advances, the WCET computation becomes more complex and it can be extremely pessimistic [18].

To help better predict and improve WCET estimates, we can leverage probabilistic systems. The policy of the cache—the bridge between the processor and the main memory to improve data access latency—has been modified, so that a random replacement policy replaces a deterministic replacement policy, such as Least-Recently-Used (LRU). The use of random replacement policy reduces occurrences of pathological cases [90] and thus the extreme cases are upper bounded with very low probabilities. Such extremely low probabilities ensure that the system execution time matches the safety requirements (e.g. 10^{-9} failure rate per hour).

Probabilistic WCETs (pWCETs) are often used as timing estimates of probabilistic systems, in which the execution time is associated with an exceedance probability. Using pWCET, one can obtain the probability for an application to exceed a certain execution time. A number of pWCET estimation techniques for random caches have been developed, and most of them have ignored the presence of faults. However, technology scaling is making both transient and permanent faults more likely [36, 50, 82]. As caches are prone to faults, they can have a significant impact on system timing behavior. Therefore, we need to take faults into consideration when computing timing estimates.

In a previous study [31], we have introduced a Static Probabilistic Timing Analysis (SPTA) method to compute pWCETs for random caches in the presence of faults. The SPTA method is based on the Markov chain model. We define cache memory layouts (i.e., combinations of main memory addresses in the cache) as system states. Each state contains a timing information vector. For each memory access, the system state and timing information is updated using a transition function. By injecting faults, a simple detection mechanism is adopted to classify the fault as transient or permanent. Experimental results show that the SPTA method can provide an accurate pWCET in presence of faults. We observe that when permanent fault rate increases, the pWCET estimates degrade greatly. To reduce permanent fault impacts on performance degradation, we can detect the permanent faults using the detection mechanism and disable the corresponding faulty block.

In this paper, we introduce transient and permanent faults to instruction caches and apply two different permanent fault detection mechanisms—rule-based detection and Dynamic Hidden Markov Model (D-HMM) detection—in different fault scenarios simulating the environmental conditions of specific aerospace applications. The choice of the two detection mechanisms aims at showing how, leveraging probabilistic models, can improve performance w.r.t purely deterministic decision making. Experimental results show that, when fault rates are high, fault detection mechanisms significantly affect the system's pWCET estimates. The rule-based detection is simpler to implement, but it provides a pWCET with more performance degradation in the high fault rate scenario. On average, rule-based detection degrades 41% for pWCET compared to the D-HMM detection mechanism.

The rest of the paper is organized as follows. Section 4.4 presents our fault model and the SPTA method for pWCET estimates in presence of faults. The two fault detection mechanisms are explained in Section 4.5. Benchmarks are evaluated in Section 6.7. Section 7.4 presents related work. Finally concluding remarks are given in Section 4.8.

4.4 Background

In this section, we summarize the SPTA method in presence of faults [31]. We first introduce the system model and the SPTA method for random caches. Then we describe probabilistic models for both the transient and permanent faults. Finally, the SPTA method for pWCET estimation in presence of faults is presented.

4.4.1 System Model

Without loss of generality, we use a fully associative cache to demonstrate the system model. The random cache we adopt has an evict-on-miss random replacement policy. When a cache miss happens, a cache block is randomly evicted and used to store the incoming data. Since the future state of the cache depends uniquely on current state, we can represent the system using a Markov chain model. The state s_i represents the memory layout of the cache. The state occurrence probability vector $\overline{S}$ and transition matrix $\overline{P}$ are defined as:

$$\overline{S} = [Pr(s_0), Pr(s_1), \cdots], \tag{4.1}$$

$$\overline{P} = \begin{pmatrix} p_{0\to0}, & p_{0\to1}, & \cdots \\ p_{1\to0}, & p_{1\to1}, & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}, \tag{4.2}$$

where $Pr(s_i)$ denotes the probability of state s_i and $p_{i\to j}$ describes the probability of going from state s_i to state s_j.

The state occurrence probability vector $\overline{S}_k$ and transition matrix $\overline{P}_k$ are used to describe the system transition at step k, and we have

$$\overline{S}_{k+1} = \overline{S}_k \overline{P}_k. \tag{4.3}$$

To account for timing information, a timing distribution variable $\mathcal{T}_i$ is assigned to each state s_i. At each step, the element $p_{i\to j}$ of $\overline{P}_k$ is computed and the timing distribution variables for all states are updated.

In our Markov chain model, the number of states increases dramatically with the number of distinct memory addresses. To reduce the computational complexity, only a fraction of the memory addresses are used to represent the states. The state addresses are changed in an adaptive way: when a memory address a_m is accessed, if it is not already included in the states, we need to select an address a according to selection rule and replace it with a_m. Thus we have $s_j = s_i \setminus \{a\} \cup \{a_m\}$. The probability and timing distribution for s_j are initialized to 0 and empty vector respectively. In addition, the probability and timing distribution for s_i are added to a pessimistic state $s_p = s_i \setminus \{a\}$.

4.4.2 Fault Model

In this paper, we consider only cache storage element faults. To obtain a safe bound, we regard each memory access as a cache miss, since the latency of a cache miss is longer than the latency of a cache hit and thus it has a higher fault occurrence probability.

For set associative caches, let n_s be the number of memory accesses between two consecutive access to the same cache set, and f be the fault probability of one cache block for one memory access, then the probability to have a fault for this cache set is computed as

$$1 - (1 - f)^{n_s}. \tag{4.4}$$

The transient faults we focus on are Single Event Upsets (SEUs) which are caused by high-

energy particles. SEUs are known to be independent events and we assume that their fault rate f_t is constant throughout the program execution and each cache block has the same fault rate.

With regard to permanent fault, we consider dynamic permanent faults, i.e. the faults caused by wear-out effects of the device. The permanent fault rate is defined as $f_p(t, T)$, where t is operation time and T is the scrubbing period, i.e. the time interval between two consecutive fault detection to avoid error accumulation. A log-normal distribution is proposed in [86] to account for cumulative wear-out effect:

$$f_p(t,T) = \frac{\mathsf{cdf}_{norm}\frac{ln(t)-\mu}{\sigma} - \mathsf{cdf}_{norm}(\frac{ln(t-T)-\mu}{\sigma})}{1-\mathsf{cdf}_{norm}(\frac{ln(t-T)-\mu}{\sigma})}, \tag{4.5}$$

where cdf_{norm} the cumulative density function of the normal distribution. The mean and standard deviation parameters of such distribution can be computed from the Mean Time To Failure (MTTF) as:

$$\begin{aligned} \mu &= ln(\frac{MTTF^2}{\sqrt{var_{MTTF} + MTTF^2}}) \\ \sigma &= \sqrt{ln(1 + \frac{var_{MTTF}}{MTTF^2})}. \end{aligned} \tag{4.6}$$

Using Equation 4.5, we can compute the permanent fault rate for a given operation time t and scrubbing period T. Note that cumulative wear-out effects are accounted for, and thus as t increases, $f(t, T)$ increases as well.

4.4.3 SPTA with Faults

When faults are injected, the states in the Markov chain model are modified accordingly. We assume that a parity check mechanism is applied so that all faults can be detected and the system can run safely.

Transient faults can be regarded as cache evictions, because if a transient fault occurs in a cache block, the data becomes invalid. When this block is accessed, the fault can be detected, and data will be fetched from the main memory. At the beginning of each memory access, new states s_p are generated from s_i due to transient faults, and they are subsets of the state s_i. The probability is calculated as $s_p \subseteq s_i$,

$$Pr(s_p) = Pr(s_i)((1-f_t)^{n_s})^{|s_p|}(1-(1-f_t)^{n_s})^{|s_i|-|s_p|} \tag{4.7}$$

where $(1-(1-f_t)^{n_s})^{|s_i|-|s_p|}$ represents the probability of cache block evictions caused by

faults and $((1-f_t)^{n_s})^{|s_p|}$ denotes the probability without evictions.

Permanent faults are different from transient faults, since the block affected by a permanent fault cannot be used for future access. To take permanent fault impacts into consideration, $N+1$ Markov chain models are created for an N-way cache. We assume that permanent faults can be detected immediately once they occur. At the beginning of each memory access, we change the states in all Markov chain models. Let s_i^m be the state with m faults, and l be the number of added permanent faults, then we have

$$Pr(s_p^{m+l}) = Pr(s_i^m)((1-f_p)^{n_s})^{N-m-l}(1-(1-f_p)^{n_s})^l \tag{4.8}$$

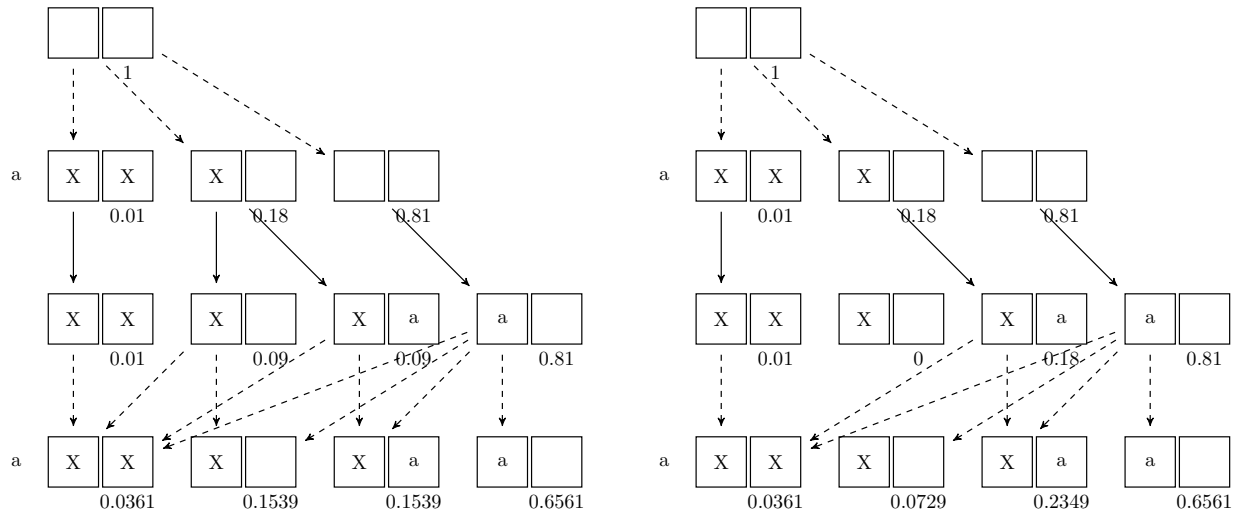
Using Equation (4.7) and (4.8), we can compute state transitions due to transient and permanent faults. The probabilities and timing distributions of the generated states are added to the existing states that have the same memory layouts. Then, the system transits in the same way as in Section 7.5.

4.5 Permanent Fault Detection Mechanisms

The SPTA method from Section 6.6 assumes that the permanent fault is detected immediately after it occurs. This can be regarded as a perfect detection. However, depending on the permanent fault detection mechanisms, the permanent fault may not be detected perfectly, i.e. there might be some delay between the classification as a permanent fault and its occurrence. As disabling the faulty block depends on its detection, this delay can affect the execution times.

Figure 4.1 is an example on how different detection mechanisms affect the system's timing behaviors. In Figure 4.1(a), the permanent fault is not detected. In Figure 4.1(b), the permanent fault is detected and the faulty block is disabled. The memory address a is accessed twice. For the second access, the hit probabilities are different: in Figure 4.1(a), it is $0.1539 + 0.6561 = 0.81$; in Figure 4.1(b), it is $0.2349 + 0.6561 = 0.891$.

We investigate two fault detection mechanisms: rule-based and Dynamic Hidden Markov Model (D-HMM) based fault detection. We assume that there is a periodic cache scrubbing, which checks the data in the cache. If any fault is detected in a cache block, the block is set as invalid. Given the fault detection mechanism, we can classify if the detected fault is transient or permanent.



(a) Faulty block not disabled.

(b) Faulty block disabled.

Figure 4.1 Impacts of different permanent fault detection mechanisms on a 2-way random cache with a permanent fault probability of 0.1 per memory access. Each block represents a state and the value at the bottom shows the probability of the state. The dotted lines denote fault occurrences and the solid lines indicate cache memory accesses. When a permanent fault occurs in a cache block, it is marked with an X.

4.5.1 Rule-based Detection

The rule-based fault detection is a simple mechanism to classify a fault as permanent. We adopt the approach in [99, 3]. We use a counter to record the number of faults of a block within a reset period. Every time a fault is detected, the counter value increases by 1. At the beginning of each reset period, the counter value is reset to 0. We can define a threshold value, such that when the counter value exceeds the threshold value, we classify the fault as permanent; otherwise the fault is regarded as transient.

4.5.2 D-HMM based Detection

To better classify permanent faults, D-HMM based detection is proposed in [86]. An HMM is defined as follows

- $P(S_0)$: an initial probability distribution of states.
- $T_{ij} = P(S_t = j|S_{t-1} = i)$: the probability of a transition from step $t-1$ to step t.
- $E_{ij} = P(O_t = j|S_t = i)$: the sensor model which shows the probability of observation given a state.

S_t	$P(O_t)$	
	Fault	No Fault
Available	0	1
SEU	1	0
Failure	1	0

Table 4.1 Sensor Model

S_{t-1}	$P(S_t)$		
	Available	SEU	Failure
Available	$1 - P(SEU \vee Failure)$	P_{SEU}	$P_{failure}(t)$
SEU	$1 - P(SEU \vee Failure)$	P_{SEU}	$P_{failure}(t)$
Failure	0	0	1

Table 4.2 D-HMM Transition Model

Using HMMs, we can predict the state of the system:

$$P_t(S = j|O_{1:t-1}) = \sum_i P_{t-1}(S = i|O_{1:t-1})T_{ij} \tag{4.9}$$

Given a new observation, the prediction can be refined as:

$$P_t(S = j|O_t = j, O_{1:t-1}) = \alpha P_t(S = i|O_{1:t-1})E_{ij} \tag{4.10}$$

where α is a normalization parameter.

The sensor model and D-HMM transition model are displayed in Table 5.3 and 5.4 respectively, where P_{SEU} is the probability of observing SEUs and $P_{failure}(t)$ is the probability of permanent fault occurrences.

We construct a D-HMM model for each cache block. At the beginning of a scrubbing phase, we update the system model using Equation (4.9). After the scrubbing phase, we apply the results to Equation 4.10. Then, we use Equation 4.9 for prediction. A threshold value is defined such that, if the predicted value exceeds the threshold value, we classify the fault as permanent.

4.6 Experimental Results

In this section, we compare different online fault detection effects on permanent faults. Mälardalen benchmarks [52] are used for evaluation. The gem5 instruction set simulator [23] is used to generate instruction traces for an ARM processor for the benchmarks. A statically

linked library and an Floating Point Unit (FPU) are adopted for the code compilation without system calls. Single-path program memory traces are generated using example inputs for all benchmarks.

We assume that the latencies for a cache hit, a cache miss and a parity check are constant, and they are set as 1 cycle, 100 cycles and 10 cycles respectively. The system is equipped with an L1 instruction cache with a size of 1024 bytes, 2-way associativity and 16-byte cache block. We consider only faults in instruction caches, but our method applies to data caches as well.

In the context of aerospace applications, the system may be exposed to different radiation environment. We consider two fault scenarios for experiments:

- Low fault rate: permanent fault with a probability of 10^{-20} per memory access, transient fault with a probability of 10^{-10} per memory access.
- High fault rate: 10^{-4} permanent fault rate, 10^{-2} transient fault rate.

We apply two random number generators for transient and permanent faults respectively and both faults are generated during simulations. Consequently, for two simulations in the same fault scenario, there may be different results due to randomness.

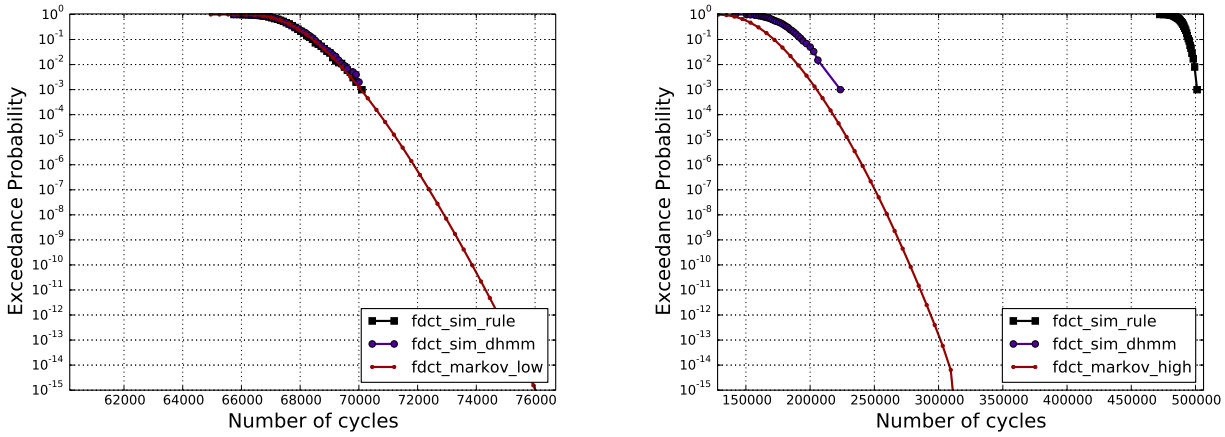
With regard to transient faults, we assume that in the low fault rate scenario, the system is protected against high-energy particles. The probability 10^{-10} is adopted, which is equivalent to $\sim 10^{-6}$ SEU bit^{-1}s^{-1}. This is the long term SEU rate for a SPENVIS device with a 4mm shield thickness on highly elliptical orbits (HEO) [108].

In the high fault rate scenario, we assume that faults happen with a probability of 10^{-2} per block, per memory access. This assumption is extremely severe and, in fact, pertaining to environmental conditions harsher than those we would find in most practical aerospace applications. It is comparable, for example, to the SEU rates that are caused by solar events in Mercury's orbit [46]. However, this "stress" test allows us to discover how the performance impacts of the rule-based and the D-HMM detection mechanisms set apart from one another in extreme conditions (despite the very short run-time of our benchmark applications).

Using Equation (4.5), we compute that for a MHz processor with 5-year MTTF, the permanent fault probability is around 10^{-20}. This is used in the low fault rate scenario. To study fault impacts in the high fault rate scenario, we increase the permanent fault rate to 10^{-4}.

In each scenario, the simulation results with the rule-based and the D-HMM based permanent fault detection mechanisms are compared. We perform 1,000 simulations for each detection mechanism. For the rule-based method, the threshold is set to 4 and the reset period is

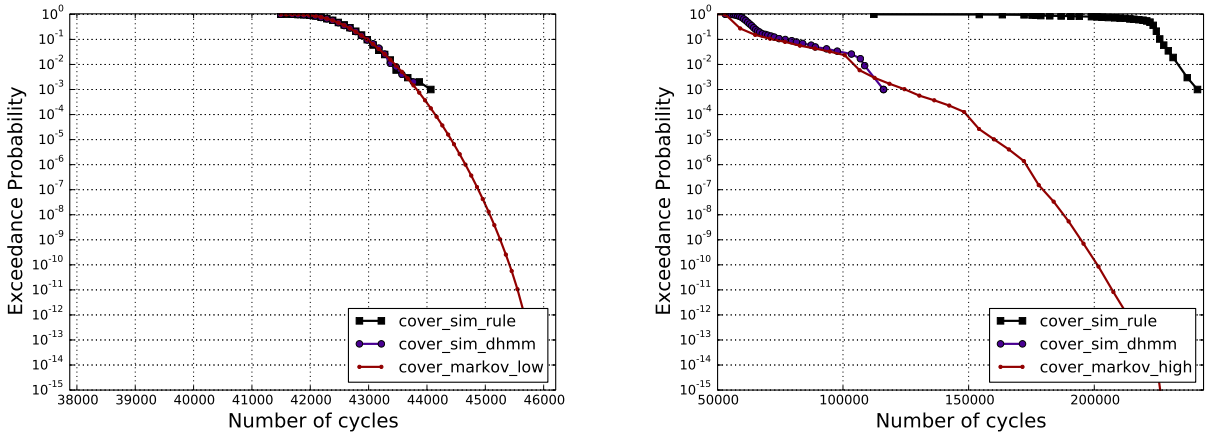
set to the execution time of of each benchmark. If 4 faults are detected, the corresponding cache block is considered to have a permanent fault and will be disabled. For D-HMM based detection, the threshold is set to 0.9. If the prediction value exceeds 0.9, the cache block will be disabled. In addition, the Markov chain based SPTA method result is displayed to show the result with perfect fault detection. Note that other thresholds produce similar results, which are not shown hereby due to limited space.



(a) *fdct* with low fault rate.

(b) *fdct* with high fault rate.

Figure 4.2 *fdct* pWCETs in low and high fault rate scenarios.



(a) *cover* with low fault rate.

(b) *cover* with high fault rate.

Figure 4.3 *cover* pWCETs in low and high fault rate scenarios.

Figure 4.2 and Figure 4.3 show results for benchmark *fdct* and *cover* respectively. The x-axis represents the number of execution cycles and y-axis the exceedance probability. In

the low fault rate scenario (Figure 4.2(a), Figure 4.3(a)), both detection mechanisms exhibit similar pWCETs, which match the result of perfect detection. In the high fault rate scenario (Figure 4.2(b), Figure 4.3(b)), for a given exceedance probability, the system execution time with rule-based permanent fault detection is much longer than that with D-HMM based permanent fault detection mechanism.

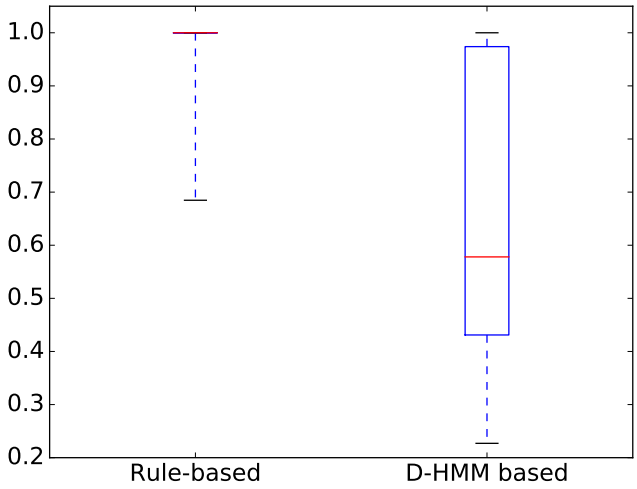


Figure 4.4 Normalized rule-based detection and D-HMM based detection execution times at the exceedance probability of 10^{-3} for all benchmarks.

The statistical results of all benchmarks in the high fault rate scenario are displayed in Figure 4.4 using box plots. The rule-based detection execution time and the D-HMM based detection execution time are normalized as

$$\tilde{X} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{4.11}$$

where X_{min} is the minimum execution time with perfect detection, rule-based detection and D-HMM based detection, X_{max} is the maximum execution time with all detection mechanisms, X is the execution time using rule-based or D-HMM based detection, and $\tilde{X}$ is the normalized value.

We observe that for rule-based detection, most execution times are normalized to 1 (only a few data points lie outsize the line-shaped box in Figure 4.4), because its execution time is usually the longest among all detection mechanisms. Using D-HMM based detection, we can improve the execution time performance. As a result, a large number of execution times are below 1 with normalization

In the low fault rate scenario, there is not much difference between pWCETs using different

permanent fault detection mechanisms, because the execution times of the benchmarks are short, and with extremely low fault rates, permanent fault rarely occurs, which limits the effect of detection mechanisms.

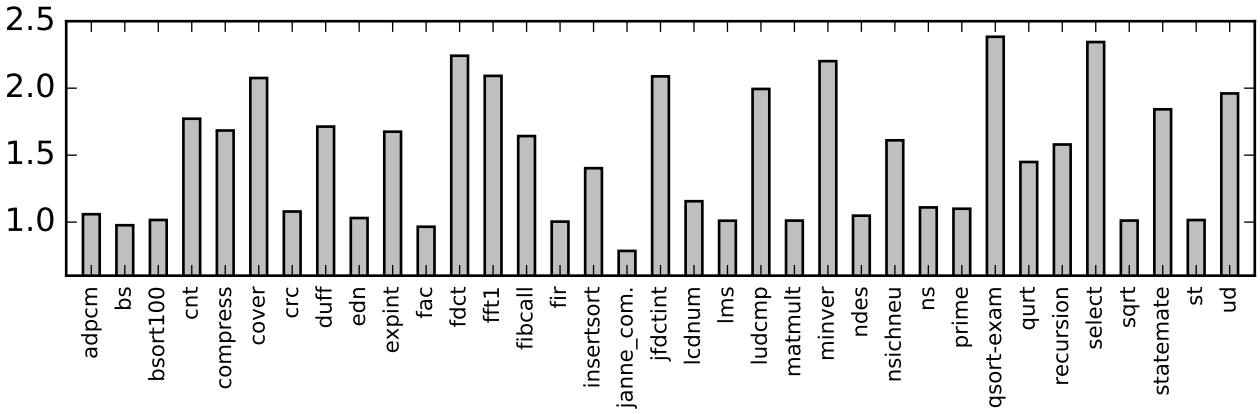


Figure 4.5 Ratio distribution in the high fault rate scenario for all benchmarks.

To show the effects of detection mechanisms in the high fault rate scenario, the detailed ratio distribution of all the benchmarks is shown in Figure 4.5. The x-axis represents different benchmarks and y-axis shows the ratio of rule-based detection execution time to D-HMM based detection execution time for each benchmark. The geometric mean is 1.41, which means on average, the rule-based detection execution time is 41% more than that using the D-HMM based detection mechanism.

4.7 Related Work

For the timing analysis of random caches, three methodologies are most commonly found in the literature: Measured Based Probabilistic Timing Analysis (MBPTA), Static Probabilistic Timing Analysis (SPTA) and the hybrid method which combines both. In this paper, we used an SPTA method which adopts a perfect fault detection mechanism. The SPTA formulae which provide fast results for single-path programs have been proposed in [118, 27, 66, 39], but their results are not as accurate as those presented here. Accurate SPTA methods are proposed in [9, 10, 49], in which cache states are used for timing analysis and the result is accurately refined at the expense of a greater computation time. [71] extends the single-path programs to multi-path programs by calculating cache state upper-bounds and execution path reductions. Previous studies have investigated the effects of fault-injection in real-time systems. [98] studies fault-tolerant systems for random caches, and estimate pWCETs using MBPTA techniques. They extend the work in [99] and study the impact of error detection, correction, diagnosis, and reconfiguration (DCDR) in different fault scenarios. [56] introduces

an SPTA method for the pWCET estimations of instruction caches. The cache under scrutiny uses an LRU replacement policy and permanent faults due to process variations are also taken into account. A fault-free pWCET and miss probability distribution are combined to obtain the pWCET with permanent faults. [55] applies reliability mechanisms to faulty instruction caches and analyze their effects, showing an improvement in pWCET estimates.

Our study focuses on transient faults and dynamic permanent faults caused by wear-out effects. In this context, we are interested in computing pWCET estimates for random replacement caches. To classify fault types, we adopt two very different online fault detection mechanisms: a deterministic, rule-based one and one based on probabilistic graphical models. This is the first study of different online permanent fault detection effects on pWCET estimations. We implement the D-HMM detection mechanism for instruction caches with faults and the results reveal that it improves pWCET estimates significantly compared to the simple rule-based detection mechanism. This can help the system meet the safety requirement.

4.8 Conclusions

In this paper, we studied the problem of classifying transient and permanent faults affecting instruction caches of real-time systems. We focus on two online permanent fault detection mechanisms —rule-based and D-HMM based detection—for random caches and we compare their effects on a system's timing analysis by analyzing traces of single-path programs. We discover that fault detection mechanisms play an important role in timing behaviors and that D-HMM detection improves pWCET estimates significantly compared to rule-based detection. In our future work, we plan to study the effects of more fault tolerant techniques. We also plan on extending the level of detail of our model to take into account the computation overhead of rule-based and D-HMM based detection to obtain even more accurate results for low exceedance probabilities.

CHAPTER 5 ARTICLE 3: PROBABILISTIC TIMING ANALYSIS OF RANDOM CACHES WITH FAULT DETECTION MECHANISMS

5.1 Preface

This article is the synthesis and extension of the work we presented in Chapter 3 [31]—a state space based technique to estimate timing behaviors using Markov Chains—and the research in Chapter 4 [30]—a comparison of rule-based detection to D-HMM based detection for random caches affected by permanent faults. This article significantly extends those contributions in the following aspects: (i) we added write-through data caches to the SPTA and permanent fault detection evaluation; (ii) we introduced the fault impact analysis into a cache contention based approach, which provides a faster yet safe probabilistic WCET (pWCET) estimate than previous approaches; (iii) we integrated the cache contention based approach with the state space approach to create a state-of-the-art SPTA method that takes faults into consideration; and (iv), in the experimental section, we added an analysis of the cache configuration impact on the pWCETs.

Authors: Chao Chen, Jacopo Panerati, and Giovanni Beltrame

Submitted to: *High Dependability Systems*, IEEE Transactions on Emerging Topics in Computing, Special Issue/Section, Third Issue of 2017 [32].

5.2 Abstract

In the real-time systems domain, random caches have been proposed as a way to simplify software timing analysis, i.e., the process of estimating the probabilistic Worst Case Execution Time (pWCET) of an application. However, in probabilistic—as well as in deterministic—systems, the technology-scaling of the cache memory manufacturing process is rendering transient and permanent faults more and more likely. These faults, in turn, affect a system's timing behavior and the complexity of its analysis. In this article, we propose a Static Probabilistic Timing Analysis (SPTA) approach for random caches that is able to account for the presence of faults—and their detection mechanisms—using a state-space modelling technique. Our experiments show that our methodology is capable of providing tight pWCET estimates. In our analysis, we compare the effects on the estimation of safe pWCET bounds of two online mechanisms for the detection and classification of faults—i.e., a rule-based system and Dynamic Hidden Markov Models (D-HMMs). The experimental results show that different mechanisms can greatly affect safe pWCET margins and that, by using D-

HMMs, the pWCET of the system can be improved with respect to rule-based detection.

5.3 Introduction

Time-critical computing systems—such as the on-board computer of a satellite—require accurate timing estimations of their software execution. If hazardous events are not handled within specific, fixed delays, the results can be catastrophic. Traditionally, designers have used extremely conservative estimations of the execution times extracted from deterministic architectural models. The major shortcoming of these approaches is the fact that they can place the Worst Case Execution Time (WCET) very far away from the actual maximum time used by the application [18].

Better, i.e., tighter, WCET estimates can greatly improve performance. To achieve this goal, one can leverage the properties of probabilistic systems. In the system under test in this work, we modified the policy of the cache—the bridge used between the processor and the main memory to improve data-access latency—so that a random replacement policy is used instead of a deterministic replacement policy such as Least-Recently-Used (LRU). Random and pseudo-random caches can now be found even in commercial processors. For example, ARM's Cortex-A73[1] and Gaisler's LEON4[2] both use random replacement policies. The use of a random cache reduces the occurrences of pathological cases—when compared to caches with a deterministic policies—-and associates extreme cases to very low probabilities, allowing for better system performance [100, 101, 90, 66]. The extremely low probabilities, in fact, ensure that the execution time matches the safety requirements (e.g., 10^{-9} failures per hour) even with tighter WCETs.

Probabilistic WCETs (pWCETs) are typically used as the timing estimates of probabilistic systems. Probabilistic WCETs associate exceedance probabilities to execution times. Thus, from a pWCET, one can also extract the probability for an application to exceed a given execution time. Several pWCET estimation techniques for random caches have been developed, however, most of them ignore or disregard the presence of faults. This is starting to appear as an oversimplifying assumption as technology scaling is making the occurrence of transient and permanent faults more and more common [36, 50, 82]. As modern caches become more prone to faults, errors can have a significant impact on a system's timing behavior. Therefore, it is crucial to take faults—and their detection/mitigation mechanisms—into consideration when computing timing estimates.

The literature accounts for a multitude of techniques that can be used to cope with transient

[1] http://infocenter.arm.com/help/index.jsp
[2] http://www.gaisler.com/index.php/products/processors/leon4

faults. For example, Reed-Solomon codes have been used extensively in space applications to detect and correct transient fault errors. In this article, we consider cache blocks that are equipped with error detection mechanisms and assume that all errors can be detected. However, differentiate transient faults from permanent failures is as non-trivial as imperative to determine the proper course of action—e.g., to fetch a memory block once a transient fault occurs.

As fault rates increase, pWCET estimates greatly degrade. When dealing with permanent faults in particular, space redundancy techniques, such as Error Correcting Codes (ECC) are not scalable because of their high cost [3]. To reduce the negative impact on performance of permanent faults, one must detect the permanently faulty cache blocks and reconfigure them so that they are not used again. The detection and classification of a fault as permanent can be outsourced to a number of ad-hoc mechanisms. Our work experiments with different fault detection mechanisms to study how they impact the pWCET estimates.

In this article, we present a Static Probabilistic Timing Analysis (SPTA) methodology for caches with a random replacement policy, that takes into consideration both transient and permanent faults. Our methodology starts from single-path program memory traces to compute probabilistic WCETs (pWCETs), i.e., exceedance probabilities associated to execution times (measured in the number of cache misses in our simulations). The approach is derived from state-space techniques [11] and it is extended in our work to consider the impact of faults. Transient and permanent fault effects are modelled as probabilistic events and implemented into the system under test through fault injection. At every memory access, faults (if they happened) are injected and the state of the system is computed. Experimental results show that our SPTA method can provide accurate pWCET estimates in the presence of faults. Moreover, we experiment with two different permanent fault detection mechanisms—rule-based detection and Dynamic Hidden Markov Model (D-HMM) based detection—in different fault scenarios that replicate the environmental conditions of actual aerospace applications. The choice of the two detection mechanisms aims at showing that, leveraging probabilistic models, one can further improve performance—with respect to deterministic decision making. Experimental results show that, when fault rates are high, the presence of fault detection mechanisms significantly affect the system's pWCET estimates. The rule-based detection is simpler to implement, but it results in greater performance degradation in the high fault rate scenario. On average, rule-based detection yields a 21% larger degradation of the pWCET compared to the D-HMM detection mechanism.

The rest of the paper is organized as follows: the related work is summarized in Section 6.4; the fault models and their impact on the system performance are presented in Section 5.5; in

Section 5.6, we describe the proposed SPTA methodology and how it handles the presence of faults; Section 5.7 details the fault detection mechanisms; experimental results on a set of real-world benchmarks are presented and discussed in Section 6.7; and finally Section 7.10 draws the concluding remarks.

5.4 Related Work

Most of the methodologies for the timing analysis of random caches that can be found in the literature fall into three main categories: (i) Measurement-based Probabilistic Timing Analysis (MBPTA); (ii) Static Probabilistic Timing Analysis (SPTA); and (iii) hybrid methods which combine the two. In this article, we propose an SPTA approach that accounts for faults and fault detection mechanisms. SPTA formulae capable of providing fast results for single-path programs have been proposed in [118, 27, 66, 39]. These methods compute the lower bound of the cache hit probability of each memory access from the cache associativity and the reuse distance (i.e., the number of memory blocks between two consecutive references to the same memory block). All hit probabilities are considered to be independent. Time Profiles (TPs) can be constructed using the miss/hit probabilities and the corresponding miss/hit latencies. Then, the pWCET can be obtained by convolving the TPs.

It is worth noting that the results obtained under the assumption of independence of all memory accesses are not as accurate as those we present here or in other works. The timing analysis methods proposed in [9, 11] exploit knowledge of the cache state and their results are typically more accurate (at the expense of a longer computational time). The ability to enumerate cache states, in particular, entails more precise pWCET estimates. The contention-based method proposed in [9] improves the pWCET results of methods that are based on the reuse distance. Selected memory block accesses are fed to a cache state enumeration-based method to obtain accurate pWCETs, while others are used in a contention-based approach to allow for a fast calculation with reasonable accuracy. The two results are then convolved to obtain an overall estimate that provides an accurate result while avoiding state explosion. Griffin *et al.* [49] proposed a different state space-based approach, which, instead, exploits lossy compression techniques to avoid the state explosion pitfall. Lesage *et al.* [71] extended the conversation from single-path programs to multi-path programs by calculating cache state upper-bounds and execution path reductions.

Previous studies have also investigated the effects of fault-injection in real-time systems. Slijepcevic *et al.* [98] studied fault-tolerant systems with random caches and estimated pWCETs using a MBPTA technique. The authors show that DTM can identify the average and worst-case performance degradation due to faults. They extend their work in [99] with a study of

the impact of error detection, correction, diagnosis, and reconfiguration (DCDR) in different fault scenarios, i.e., low-, medium- and high-error scenarios for both transient and permanent faults. Their results show that, in most cases, pWCET estimates are negligibly affected by the presence of faults. In some cases, programs are more sensitive to the number of available cache blocks, which exacerbates the negative effect of permanent faults.

Hardy and Puaut [56] present an SPTA method for the pWCET estimation of instruction caches. The research work is focused on caches using an LRU replacement policy and it takes into account those permanent faults that are due to process variations. The fault-free pWCET and the probability distribution of cache misses are combined to obtain the pWCET with permanent faults. Hardy *et al.* [55] also investigated the introduction of reliability mechanisms into instruction caches with permanent faults. Two mechanisms are evaluated—one extra fixed way per set and a shared buffer. By using the SPTA techniques from [56], it is shown that simple reliability mechanisms can mitigate the impact of faulty cache blocks and considerably improve pWCETs.

The uniqueness of our study resides in how it considers both transient faults and dynamic permanent faults caused by wear-out effects. In this particular context, we are interested in computing the pWCET estimates of caches with random replacement policies. The SPTA method we propose can provide tight pWCET estimates while taking into account the occurrence of faults. To classify faults (as transient or permanent), we adopt two different online fault detection mechanisms: a deterministic, rule-based approach and a mechanism based on probabilistic graphical models (D-HMMs). This is the first study of the effects of different online permanent fault detection mechanisms on pWCET estimations dealing with both instruction and data caches. Having implemented the D-HMM detection mechanism in a random replacement cache injected with faults, we discover that it significantly improves the pWCET estimates when compared to the static rule-based detection mechanism. This, in turn, offers a fundamental advantage in helping the system to meet its safety requirements.

5.5 The Impact of Faults

This section is dedicated to the description of both the transient and permanent fault models on which this work is based. These models are then used for the injection of faults into the system under test. In our model, we only consider those faults that occur in the storage elements of the cache. Faults happening in the combinational circuit entail a range of different behaviours (e.g., transient faults have no impact on future operations) and they can be detected using additional mechanisms that may increase the time to access cache, but their study is beyond of the scope of this work. In our analysis, we compute fault occurrence

probabilities for each memory access. Since a cache miss takes longer than a hit, to simplify the analysis and obtain a safe bound, in following sections, we use the latency of a cache miss for the fault probability calculations of each memory access.

5.5.1 Transient Faults

A Single Event Upset (SEU) is a change of state in a memory element, usually caused by the impact of a high-energy particle. SEUs induce transient faults since they do not cause permanent damage: the memory elements they affected can still be correctly used after the successive write operation. SEUs due to space radiation are known to be independent, and we assume they are uniformly distributed in space, i.e., each cache block has the same fault probability throughout the program execution. We thus define the transient fault probability for each memory access as f_t.

When a transient fault occurs, a cache block remains faulty until the fault is detected (and corrected). This is because transient faults happen in storage elements that do not recover autonomously. As a result, transient faults can have lasting effects. After the fault is detected, the content of the block is labelled as invalid. However, this does not affect the following data storage operations on the same block.

To cope with transient faults, many techniques have been proposed. For example, Reed-Solomon codes have been used extensively in space applications to detect and correct transient fault errors. In our work, we employ a classic parity check fault detection mechanism—where parity bits are are added to the data—to the first level (L1) cache. Our choice is motivated by the smaller area requirements and lower delay penalties of parity checking when compared to other commonly used single error correction-double error detection (SEC-DED) techniques [79].

When a cache block is accessed, the stored parity bits are examined to discover whether a transient fault has occurred. At relatively low fault rates, we assume that almost all faults can be detected. If a fault is detected from the result of parity checking, the corresponding block data is labelled as invalid and fetched from the main memory again. If no fault is detected and the access is a cache hit, the data in the cache is used as is.

5.5.2 Permanent Faults

Permanent faults are those faults whose effects last from the moment they appear until the end of the system's life cylce. When a permanent fault occurs, the affected cache block cannot be re-used. In this work, we only deal with the dynamically arising permanent faults

induced by the wear-out of components. These faults can occur at any moment during the execution of a program.

To model permanent faults we start by defining the probability $f_p(t,T)$ of a permanent fault to occur. This is the probability of a fault in a system component by time t, i.e., $failure \leq t$, given that the component was still functional at the end of the previous interval $t-T$, i.e., $failure > t-T$. T is the scrubbing period, i.e., the time interval between two consecutive fault detection operations. Scrubbing happens periodically to avoid the accumulation of error. This probability can be computed from the formula in [86] using Kolmogorov's definition as:

$$\begin{aligned} f_p(t,T) &= Pr(failure \leq t | failure > t-T) \\ &= \frac{Pr(failure \leq t \wedge failure > t-T)}{Pr(failure > t-T)} \\ &= \frac{\mathsf{cdf}_{failure}(t) - \mathsf{cdf}_{failure}(t-T)}{1-\mathsf{cdf}_{failure}(t-T)}, \end{aligned} \tag{5.1}$$

with $\mathsf{cdf}_{failure}$ being the cumulative density function of the random variable failure describing the time at which the failure occurs.

In the literature on reliability, several probability distributions are used to model failure times [86]. A recurring one is the exponential distribution; however, the exponential representation is somewhat imprecise as it lacks the ability to capture the increasing failure probability due to the accumulated wear of a component. A good alternative capable of overcoming this limitation is the log-normal failure distribution:

$$f_p(t,T) = \frac{\mathsf{cdf}_{norm}\left(\frac{ln(t)-\mu}{\sigma}\right) - \mathsf{cdf}_{norm}\left(\frac{ln(t-T)-\mu}{\sigma}\right)}{1-\mathsf{cdf}_{norm}\left(\frac{ln(t-T)-\mu}{\sigma}\right)}, \tag{5.2}$$

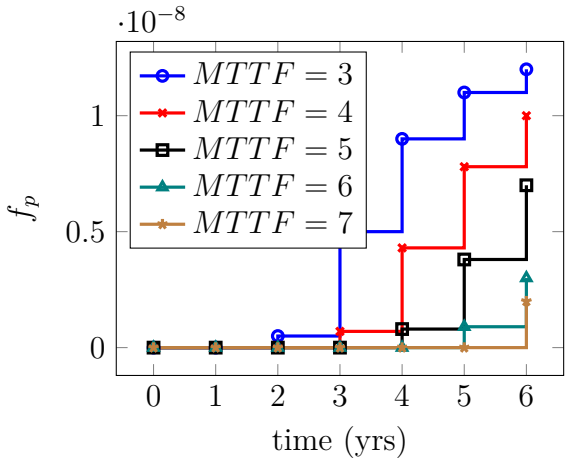
where cdf_{norm} is the cumulative density function of the normal distribution. The mean and standard deviation parameters of this distribution can be computed from a device's Mean Time To Failure (MTTF) as:

$$\begin{aligned} \mu &= ln\left(\frac{MTTF^2}{\sqrt{var_{MTTF} + MTTF^2}}\right) \\ \sigma &= \sqrt{ln\left(1+\frac{var_{MTTF}}{MTTF^2}\right)}. \end{aligned} \tag{5.3}$$

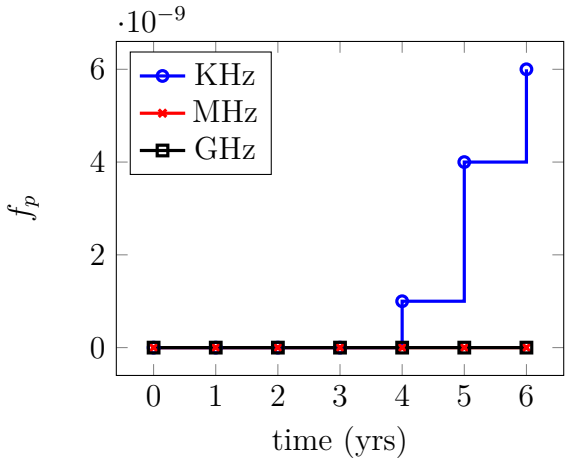
Note that, in Equation 5.2, $f_p(t,T)$ depends on the actual time t as well as the scrubbing period T. A non-memoryless distribution function, in fact, can describe the occurrence of a failures with detail that cannot be grasped by a memory-less distribution (e.g., the

exponential distribution) [45].

Figure 5.1(a) summarizes the comparison of the log-normally distributed failure times with different MTTFs. The plot is discretized in years. We can see that, as the MTTF increases, the permanent fault probability f_p for each memory access decreases, because a smaller fault probability is induced by the longer lifetime. In addition, f_p is a monotonic function of time. As the system's operation time increases, f_p increases continuously. In this we, however, we consider a constant f_p, because the benchmarks we employ for the WCET evaluations tend to be relatively short [52]. From Equation (5.2), we can see that when the execution times of our benchmarks are sufficiently short, f_p's growth is negligible.



(a) Increasing the MTTF (in years) the probability of a failure by any given time decreases consistently.



(b) Varying the system frequency, the probability of failure for ∼MHz and ∼GHz operations are, respectively, 10^{-3} and 10^{-6} smaller than those for ∼KHz.

Figure 5.1 Failure probability f_p plotted as a function of the MTTF and the platform frequency.

Figure 5.1(b) presents the comparison of the log-normally distributed failure times at different operating frequencies. The MTTF is assumed to be 5 years. At first, the fault probability per memory access is always extremely low, e.g., at a ∼KHz operating frequency, the permanent rate is 4.4×10^{-18} at year 2. The fault probabilities at ∼MHz and ∼GHz are 10^{-3} and 10^{-6} smaller respectively.

5.6 SPTA Methodology

This section starts with the introduction of a cache model with an evict-on-miss random replacement policy. Then, it proceeds to describe a state-of-the-art SPTA method for ran-

dom instruction caches which comprises two separate approaches: a cache contention based approach and a state space based approach. For each of these two methods, we study the impact of faults and discuss how we augmented the SPTA methodology to account for it. A major advantage of the joint use of the contention based approach and the state space based approach is the ability to keep computational complexity within the tractable realm.

The SPTA method under scrutiny is directly applicable to fully associative caches and it can be generalized to set associative caches (by separately performing the analysis of each cache set as if it was a fully associative cache). We focus on single-path programs and assumes no pre-emption. In addition, cache miss and cache hit latencies are assumed to be constant, to avoid timing anomalies.

5.6.1 The Cache Model

An abstraction of a set-associative cache is shown in Figure 5.2. This type of cache is divided into multiple sets and, for each of them, it provides a number of ways to store cache blocks. Each memory address used by the cache is broken into three parts: tag, set and offset. The offset locates the data within each block and the set is used to find which cache block should be used with a given memory address. As multiple memory addresses can be stored in the same cache block, the tag is stored within each cache block for comparison, to identify the actual memory address it refers to.

	Way 1	Way 2	Way 3	Way 4
Set 0		0x...		
Set 1			0x...	0x...
Set 2				
Set 3	0x...			

Figure 5.2 Set-associative cache representation.

In a cache with an evict-on-miss random replacement policy, every time a cache miss happens, a cache block in the same set is randomly selected and replaced with the new data (the term *data* is used, here and in the following, to refer to the content of a memory address).

Our method can be applied to both instruction and data caches. In data caches, there are two choices for the writing policy: a write-through or a write-back policies. A write-back policy writes the data temporarily in the cache, and then transfers the data to the main

memory when the cache block is evicted. However, in the presence of faults, the data in the cache may be corrupted, resulting in data consistency issues. Hence, in this paper, we consider data caches with a write-through policy, which is often combined with a *no write allocate* policy, i.e., on a write miss, no cache line is allocated as shown in Figure 5.3.

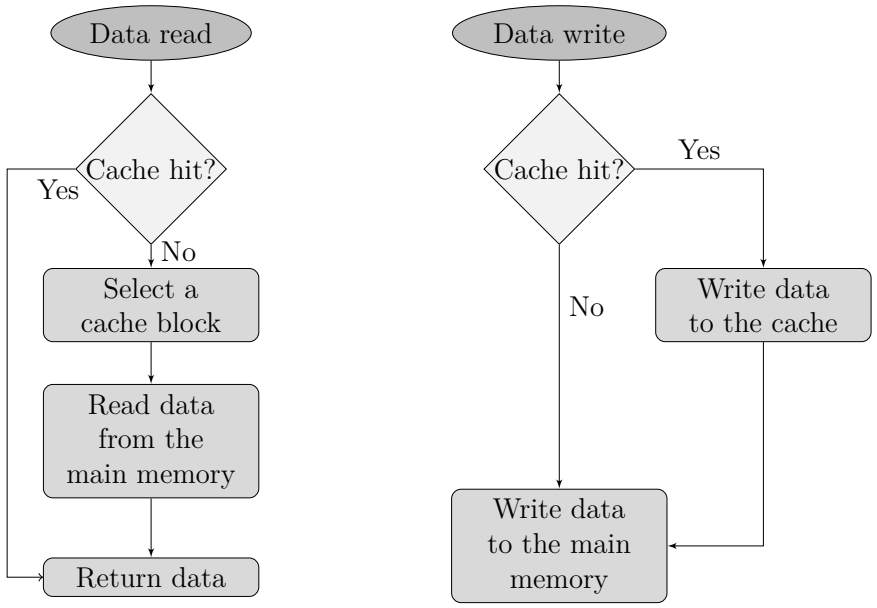


Figure 5.3 A write-through data cache with *no write allocate*.

Note that each memory address has one and only one corresponding cache set. As a result, cache blocks belonging to separate sets cannot affect one another, and results relative to different cache sets can be regarded as independent variables. For this reason, we can use SPTA to calculate the pWCET of each set and then obtain the overall pWCET by convolving the pWCETs of all cache sets [27].

5.6.2 Cache Contention Approach

To perform SPTA on instruction caches, we exploit a trace $T = [e_1, ...e_n]$, where each e_i represents a memory block. The reuse distance rd is defined as the number of memory blocks accessed between two consecutive references to the same memory block, and it is calculated as:

$$rd(e_i, T) = \begin{cases} i - j - 1 & \text{if } \exists e_j \in T : e_j = e_i, \\ & \forall p : j < p < i, e_p \neq e_i \\ \infty & \text{otherwise} \end{cases} \tag{5.4}$$

Cache contention $con(e_i, T)$ is a metric proposed in [11] to help calculate cache hit probabilities and it includes all the cache blocks that may contribute to cache hits. It is defined as:

$$con(e_i, T) = \begin{cases} \infty & \text{if } rd(e_i, T) = \infty \\ |conS(e_i, T)| & \text{otherwise} \end{cases} \tag{5.5}$$

$$\begin{aligned} conS(e_i, T) = \{e_j \in T | i - rd(e_i, T) < j < i \wedge \hat{P}(e_j^{hit}) \neq 0\} \\ \cup \{e_r \in T | r = i - rd(e_i, T)\} \end{aligned} \tag{5.6}$$

The lower bound of the actual hit probability $\hat{P}^N(e_i^{hit})$ for a cache with associativity N is computed as:

$$\hat{P}^N(e_i^{hit}) = \begin{cases} 0 & con(e_i, T) \geq N \\ \left(\frac{N-1}{N}\right)^{rd(e_i,T)} & \text{otherwise} \end{cases} \tag{5.7}$$

5.6.3 Extending the Cache Contention Approach

In Figure 5.3, we can see that reading data is akin to reading instructions. For what concerns writing data, the write-through policy always writes data directly to the main memory. Consequently, one can regard each write operation as a cache miss. We introduce the binary function T to express whether a memory request is a cache read or a cache write:

$$T(e_i) = \begin{cases} \mathcal{R} & \text{instruction or data read} \\ \mathcal{W} & \text{data write} \end{cases} \tag{5.8}$$

We first tackle the case of a cache read, i.e., $T(e_i) = \mathcal{R}$. To account for the impact of faults, we need to modify $\hat{P}^N(e_i^{hit})$ accordingly. If a permanent fault has been detected, the available cache block number may be smaller than the associativity N. Consequently, we use N groups of hit probabilities for our analysis and use $k \in \mathbb{N}, 0 \leq k \leq N$ to represent k available cache blocks. We define $\hat{P}_F^k(e_i^{hit})$ as the lower bound of the hit probability for the kth group in the presence of faults, and we calculate it as follows:

$$\hat{P}_F^k(e_i^{hit}) = \prod_{j=m}^{i} a_t^j \cdot \prod_{j=m}^{i} a_p^j \cdot b_k^i \cdot \hat{P}^k(e_i^{hit}). \tag{5.9}$$

Knowing that m is the memory index such that $m < i \wedge e_{m-1} = e_i \wedge \forall m < p < i : e_p \neq e_i$.

We use a_t^i to calculate the transient faults effects as it denotes the probability of not observing a transient fault when accessing memory block e_i. When a transient fault occurs in the cache, we assume that it is detected and a cache miss ensues. Let n_a^i be the number of memory accesses between accessing e_{i-1} and e_i. We have $n_a^i = 1$ if e_{i-1} and e_i are in the same cache set; otherwise it is a value > 1 because different cache sets are accessed.

Since the transient fault probability for each memory access is f_t, we have that:

$$a_t^i = (1 - f_t)^{n_a^i} \tag{5.10}$$

Let a_p^i be the probability of not encountering a permanent fault when accessing e_i. Similarly to what we did with a_t^i, we can calculate a_p^i from the permanent fault probability f_p as:

$$a_p^i = (1 - f_p)^{n_a^i} \tag{5.11}$$

With a_t^i, a_p^i and m, we derive the terms $\prod_{j=m}^{i} a_t^i$ and $\prod_{j=m}^{i} a_p^i$, which account for the hit probability reductions due to transient faults and permanent faults, respectively.

We define d_k^i as the occurrence probability of the k-th group after having accessed memory block e_{i-1}. This includes the accumulated effects of permanent faults and it is computed as follows:

$$d_k^i = \sum_{j=k}^{N} \left(\begin{array}{c} j \\ k \end{array} \right) (1 - a_p^i)^{j-k} (a_p^i)^k d_j^{i-1} \tag{5.12}$$

Knowing that:

$$d_k^0 = \begin{cases} 1 & \text{if } k = N \\ 0 & \text{otherwise} \end{cases} \tag{5.13}$$

Then, b_k^i is computed as follows:

$$b_k^i = \begin{cases} d_k^j & \text{if } \exists e_j \in T : e_j = e_i, \\ & \forall p : j < p < i, e_p \neq e_i \\ 1 & \text{otherwise} \end{cases} \tag{5.14}$$

Let $\hat{P}_F(e_i^{hit})$ be the hit probability with faults for both cache read and write operations. For a cache read, this is equal to the sum of the hit probabilities of all N groups reading the

cache. In the case of a cache write, instead, we have a cache miss. Thus, we have that:

$$\hat{P}_F(e_i^{hit}) = \begin{cases} \sum_{j=1}^{N} \hat{P}_F^j(e_i^{hit}) & \text{if } T(e_i) = \mathcal{R} \\ 0 & \text{if } T(e_i) = \mathcal{W} \end{cases} \tag{5.15}$$

Table 5.1 presents an example of the calculations required to compute the hit probabilities with fault impacts when reading the memory sequence a, b, a.

Table 5.1 Calculation example when reading memory sequence a, b, a through a fully associative cache with associativity $N = 2$. Transient and permanent fault rates are $f_t = 0.2$ and $f_p = 0.1$, respectively, yielding $a_t^i = 0.8$ and $a_p^i = 0.9$.

e_i	a	b	a
rd	∞	∞	1
con	∞	∞	1
$\hat{P}^N$	0	0	0.5
d_1^i	0.18	0.3078	0.3951
d_2^i	0.81	0.6561	0.5314
b_1^i	1	1	0.18
b_2^i	1	1	0.81
$\hat{P}_F^1$	0	0	0.04666
$\hat{P}_F^2$	0	0	0.21
$\hat{P}_F$	0	0	0.2567

5.6.4 State Space Approach

Contention based approaches provide a way to perform fast SPTA on random caches. However, their results can be imprecise. To improve their accuracy, Altmeyer *et al.* [11] use a state space based approach.

In a state space based approach, a cache state is defined as $CS = (E, P, D)$, where $E \subseteq \mathbb{E}$ is a set of memory blocks in the cache, $P \subseteq \mathbb{R}$ is the occurrence probability and $D : \mathbb{N} \to \mathbb{R}$ is the cache miss distribution. To keep this approach within tractable complexity, only one memory block set R is selected for thorough analysis. The other blocks are regarded as cache evictions and analyzed using a contention based approach, which produces a much fast result for reasonable accuracy.

Assuming that the cache is initially empty, we use $\perp$ to represent an empty cache block or a non-selected memory block $e \notin R$. Then the initial state space is defined as $CS_{init} = (\{\perp$

, ... $\perp\}, 1, D)$, with:

$$D(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.16}$$

When a memory block e is accessed, the update function u is used to model how the state space changes. If the memory block is in the cache, the state is unaffected. Otherwise, either a cache block will be evicted or a new cache block will be used for the incoming memory block. The update function is defined as:

$$u((E, P, D), e) = \begin{cases} \{(E, P, D)\} & \text{if } e \in R \wedge e \in E \\ miss((E, P, D), e) & \text{otherwise} \end{cases} \tag{5.17}$$

Knowing that:

$$miss((E, P, D), e) = \{(E \setminus e') \cup \{e\}, P \cdot \frac{1}{N}, D' | e' \in E\} \tag{5.18}$$

and

$$D'(x) = \begin{cases} D(x) & \text{if } e \notin R \\ 0 & \text{if } e \in R \wedge x = 0 \\ D(x-1) & \text{otherwise} \end{cases} \tag{5.19}$$

To reduce the number of states, the merge operation $\uplus$ can be used to join two states with the same memory blocks. This operation is defined as:

$$(E_1, P_1, D_1) \uplus (E_2, P_2, D_2) = \\ \begin{cases} \left\{\left(E_1, P_1 + P_2, \left(\frac{P_1}{P_1+P_2} D_1\right) \oplus \left(\frac{P_2}{P_1+P_2} D_2\right)\right)\right\} & E_1 = E_2 \\ \{(E_1, P_1, D_1), (E_2, P_2, D_2)\} & \text{otherwise} \end{cases} \tag{5.20}$$

where $\oplus$ is the summation of the distributions—i.e., $(D_1 + D_2)(x) = D_1(x) + D_2(x)$)—and $p \cdot D$ is the multiplication of each element in D by p.

From the initial state space and the update function, one can calculate how a distribution varies with the incoming memory block addresses. The final distribution is then computed as the weighted sum of all distributions.

5.6.5 Extending the State Space Approach

The state space approach that we just presented applies to read operations. We need to extend it to the write operations of write-through caches. Every data write operation is

regarded as a cache miss. Therefore, in the case of a data write, we have the update function u:

$$u((E,P,D),e) = \{(E,P,D')\} \quad \text{if} \quad T(e) = \mathcal{W} \tag{5.21}$$

To take transient and permanent faults into consideration, we need to modify the update function, so that also the transient and permanent fault detection mechanisms are accounted for. To do so, we use the method proposed in [31].

We define the transient fault impact function tf as follows:

$$tf((E,P,D)) = \{(E \setminus E', P \cdot (a_t^i)^{|E \setminus E'|} \cdot (1-a_t^i)^{|E'|}, D) | E' \subseteq E\} \tag{5.22}$$

When a transient fault occurs, we assumed that it can be detected. As a consequence, a transient fault can be regarded as a cache eviction, i.e., the data in the cache block become invalid and a cache miss occurs. Every cache block may be affected by a transient fault at any time, and the probability for a cache block of not being interested by a transient fault is calculated using Equation (5.10). The cache blocks are partitioned into $E \setminus E'$ (i.e., the blocks without transient faults) and those that are evicted E' (i.e., the blocks with transient faults).

Besides transient faults, we also need to further modify our methodology to deal with permanent faults. Since permanent faults can reduce the number of available cache blocks for a cache set, we use S_k to represent the cache state set with $k \in \mathbb{N}, 0 \leq k \leq N$ available blocks. Thus, we have $CS \in S_k$.

We define the permanent fault impact function as pf. Suppose that the state before a permanent fault occurs is $CS = (E,P,D) \in S_i$ and the state with permanent faults is $CS' \in S_j, j \leq i$, i.e., the number of available cache blocks decreases because of the occurrence of permanent faults. Then, we compute pf as follows:

$$\begin{aligned} pf((E,P,D)) = (E \setminus E', P \cdot (a_p^i)^{|E \setminus E'|} \cdot (1-a_p^i)^{|E'|}, D) \\ |E' \subseteq E, CS' \in S_{|E \setminus E'|} \end{aligned} \tag{5.23}$$

The permanent fault impact function describes how a permanent fault affects states in different cache sets. Note that when we use a state $CS \in S_i$, we consider that the cache associativity is i (i.e., cache blocks with permanent faults are detected and disabled). For now, we ignore the impacts of a permanent fault detection mechanism and a perfect detection mechanism is assumed, i.e., the permanent faults are detected immediately after they occur.

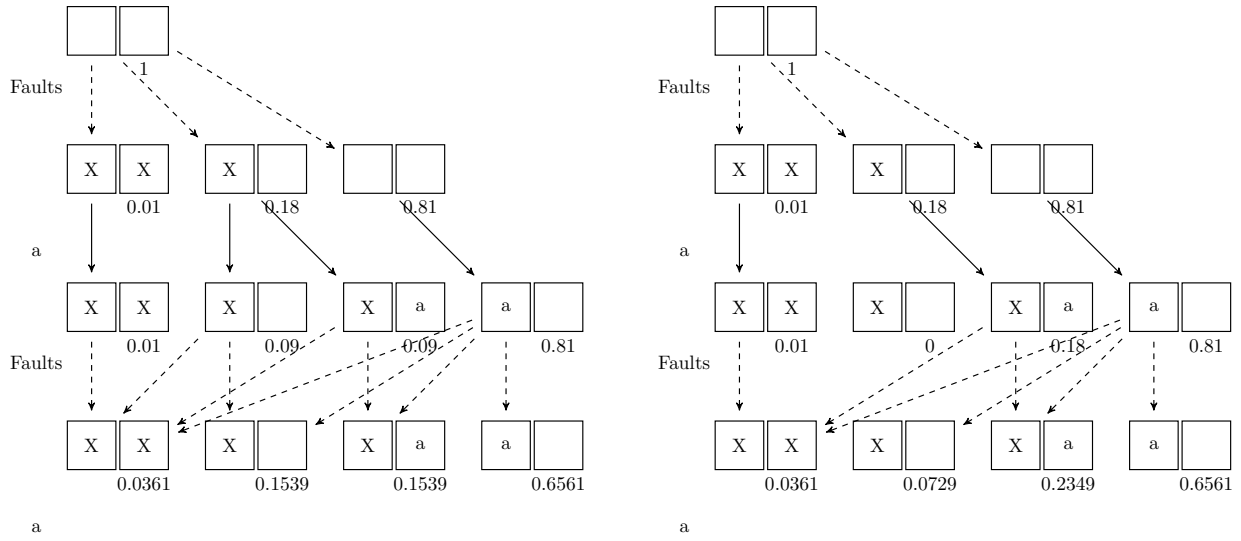
Equations (5.22) and (5.23) describe transient and permanent fault impacts, respectively. We apply them to the cache states before the update function u in order to perform SPTA estimates in the presence of faults. Table 5.2 details the SPTA in the presence of faults using the same scenario of Table 5.1.

Table 5.2 Calculation example for the state space based approach when reading the memory sequence a, b, a with the same cache configuration and fault rates of Table 5.1. At the beginning of each memory access, the fault impacts are evaluated using the transient fault impact function tf and the permanent fault impact function pf. Finally, the update function u is used to update the cache states.

Initial States	$(\{\perp, \perp\}, 1, [1])$
Transient Faults	$(\{\perp, \perp\}, 1, [1])$
Permanent Faults	$(\{\perp, \perp\}, 0.81, [1]), (\{\perp\}, 0.18, [1])$ $(\{\}, 0.01, [1])$
a	$(\{a, \perp\}, 0.81, [0, 1]), (\{a\}, 0.18, [0, 1])$ $(\{\}, 0.01, [0, 1])$
Transient Faults	$(\{a, \perp\}, 0.648, [0, 1]), (\{\perp, \perp\}, 0.162, [0, 1])$ $(\{a\}, 0.144, [0, 1]), (\{\perp\}, 0.036, [0, 1]),$ $(\{\}, 0.01, [0, 1])$
Permanent Faults	$(\{a, \perp\}, 0.5249, [0, 1]), (\{\perp, \perp\}, 0.1312, [0, 1])$ $(\{a\}, 0.1879, [0, 1]), (\{\perp\}, 0.1199, [0, 1]),$ $(\{\}, 0.0361, [0, 1])$
b	$(\{a, b\}, 0.2625, [0, 0, 1]), (\{b, \perp\}, 0.3937, [0, 0, 1])$ $(\{b\}, 0.3078, [0, 0, 1]), (\{\}, 0.0361, [0, 0, 1])$
Transient Faults	$(\{a, b\}, 0.168, [0, 0, 1]), (\{a, \perp\}, 0.042, [0, 0, 1])$ $(\{b, \perp\}, 0.357, [0, 0, 1]), (\{\perp, \perp\}, 0.08924, [0, 0, 1])$ $(\{b\}, 0.2462, [0, 0, 1]), (\{\perp\}, 0.06156, [0, 0, 1])$ $(\{\}, 0.0361, [0, 1])$
Permanent Faults	$(\{a, b\}, 0.1361, [0, 0, 1]), (\{a, \perp\}, 0.03402, [0, 0, 1])$ $(\{b, \perp\}, 0.2892, [0, 0, 1]), (\{\perp, \perp\}, 0.07228, [0, 0, 1])$ $(\{a\}, 0.0189, [0, 0, 1]), (\{b\}, 0.2688, [0, 0, 1])$ $(\{\perp\}, 0.1074, [0, 0, 1]), (\{\}, 0.07344, [0, 0, 1])$
a	$(\{a, b\}, 0.2807, [0, 0, 0.4849, 0.5151])$ $(\{a, \perp\}, 0.2509, [0, 0, 0.1356, 0.8644])$ $(\{a\}, 0.3951, [0, 0, 0.04784, 0.9522])$ $(\{\}, 0.07344, [0, 0, 0, 1])$

5.7 Permanent Fault Detection Mechanisms

The SPTA method presented in Section 5.6 assumes that permanent faults are detected—and addressed—immediately after they occur. This assumption can be regarded as one of "perfect detection". However, in the real world, a permanent fault is not detected perfectly/immediately. Depending on the applied permanent fault detection mechanism, since the fault occurs, there might be a longer or shorter delay before a is is identified/classified as permanent. Because disabling the faulty block depends on its detection, this delay can affect the execution times.



(a) The faulty block is not disabled.

(b) The faulty block is disabled.

Figure 5.4 The impact of permanent fault detection on a 2-way random cache with a permanent fault probability of 0.1 per memory access. One square represents a cache block whose status can be: a) empty; b) containing a memory block; or c) permanently faulty. The value at the bottom of the block indicates the probability of the status. The dotted lines denote the behavior of the cache blocks in the presence of permanent faults and the solid lines indicate the behavior of a memory access. When a permanent fault occurs in a cache block, it is marked with an X.

Figure 5.4 shows how the performance of detection mechanisms affect the system's timing behavior for the example memory access sequence a, a. The second access of a is not always a cache hit, because a permanent fault can make a cache block invalid. In Subfigure 5.4(a), the permanent fault is not detected. In Subfigure 5.4(b), the permanent fault is detected and the faulty block is disabled. We can see that when a is stored in the cache, the probabilities of the cache block status are different, since a different number of cache blocks is used. For the

second access of a, the hit probabilities can be computed by summing up the probabilities of all cache blocks that contain a as follows: in Subfigure 5.4(a), $0.1539 + 0.6561 = 0.81$; in Figure 5.4(b), $0.2349 + 0.6561 = 0.891$. The difference arises from the presence or absence of prompt fault detection.

In this article, we investigate two permanent fault detection mechanisms from [30]: rule-based and Dynamic Hidden Markov Model (D-HMM) based fault classification. We assume that cache scrubbing happens periodically to check for the onset of errors in the data. If any fault is detected in a cache block, the block is labelled as invalid. Using a fault classification mechanism, we can tell apart transient and permanent faults—and detect those in the latter category.

5.7.1 Rule-based Detection

The rule-based fault detection is a straightforward mechanism to classify a fault as permanent. We adopt the approach presented in [99, 3]. We use a counter to record the number of faults in a block within a reset period. Every time a fault is detected, the counter value increases by 1.

At the beginning of each reset period, the counter value is set to 0. We can then define a threshold value such that, when the counter value exceeds it, we classify the fault as permanent; otherwise the fault is regarded as transient.

5.7.2 D-HMM based Detection

To more reliably classify permanent faults amid transient faults, a D-HMM based detection mechanism was proposed by Panerati *et al.* [86]. HMMs are defined by the following components:

- $P(S_0)$: the initial probability distribution over the domain of the state variable.
- $T_{ij} = P(S_t = j | S_{t-1} = i)$: a one time-step probabilistic transition function for the state variable.
- $E_{ij} = P(O_t = j | S_t = i)$: the probabilistic sensor model revealing the probability of every observation in any given a state.

Table 5.3 D-HMM sensor model, "Available" means that there is no fault in the block. Otherwise a permanent failure or transient SEU have occurred.

S_t	$P(O_t\|S_t)$	
	Fault	No Fault
Available	0	1
SEU	1	0
Failure	1	0

Table 5.4 D-HMM transition model, SEUs and permanent failures are treated as independent from one another.

S_{t-1}	$P(S_t\|S_{t-1})$		
	Available	SEU	Failure
Available	$1 - P(SEU \vee Failure)$	P_{SEU}	$P_{failure}(t)$
SEU	$1 - P(SEU \vee Failure)$	P_{SEU}	$P_{failure}(t)$
Failure	0	0	1

One of the advantages of the HMM representation is the fact that we can predict the state variable with linear time complexity:

$$P_t(S = j|O_{1:t-1}) = \sum_i P_{t-1}(S = i|O_{1:t-1})T_{ij} \tag{5.24}$$

Given a new observation, the prediction can also be refined as:

$$P_t(S = j|O_t = j, O_{1:t-1}) = \alpha P_t(S = i|O_{1:t-1})E_{ij} \tag{5.25}$$

where α is a normalization parameter.

The sensor model and transition model used by the D-HMM classifier are displayed in Table 5.3 and 5.4, respectively. In these models, P_{SEU} is the probability of a SEU and $P_{failure}(t)$ is the probability of a permanent fault to occur during the scrubbing period at time t.

We initialize a D-HMM model for each cache block. At the beginning of a scrubbing phase, we update the system model using Equation (5.24). After the scrubbing phase, we refine the result with Equation 5.25. Then, we re-apply Equation 5.24 for prediction. A "belief" threshold value is chosen so that, if the predicted value (for the likelihood of the block to be in the "failure" state) exceeds it, we classify the fault as permanent.

5.8 Evaluation

In this section, we present the experimental setup and evaluation of the proposed SPTA methodology in the presence of both permanent and transient faults. Furthermore, we compare the different online fault detection mechanisms from the previous section to study their impact on the overall performance.

5.8.1 Experimental Setup

For our evaluation, we choose to use the Mälardalen benchmark suite [52]. These are very popular benchmarks for the WCET estimation problem. To generate our input, we use the gem5 instruction set simulator [23] to produce instruction and data traces for an ARM processor. The data traces are not used in the experiments. A statically linked library is adopted for the compilation of the benchmark executables.

System Under Test

We consider a computing system provided with a Floating Point Unit (FPU). For all benchmarks, single-path program memory traces are generated using the pre-defined inputs. The system is equipped with separate L1 instruction and data caches. In the following, several cache configurations are evaluated.

We assume that the latencies for a cache hit, a cache miss and error detection overhead for each memory access are constant, and they are C_h, C_m, and C_d, respectively. The total number of cache accesses, cache misses and cache hits are denoted as as N_t, n_m and n_h. Thus the execution time t is calculated as

$$t = n_m \cdot (C_m + C_d) + (N_t - n_h) \cdot (C_h + C_d). \tag{5.26}$$

We observe that t is a linear function of n_m. Thus, we can use the number of misses as a metric to denote the overall execution time.

Fault Scenarios

With regard to fault scenarios, we setup a low and a high fault scenario for our experiments. In each scenario, each cache block has a transient fault probability and a permanent fault probability on each cache access. With regard to transient faults, we assume that in the low fault rate scenario, the system is protected against high-energy particles. We use a probability value of $1e^{-10}$ which is equivalent to $\sim 1e^{-6}$ SEU bit^{-1}s^{-1}. This is, for example,

the long term SEU rate for a device with a 4mm shield thickness on a highly elliptical orbit (HEO) [108] as computed by ESA's SPENVIS (SPace ENVironment Information System).

In the high fault rate scenario, we assume that faults happen with a probability of $1e^{-2}$ per block, per memory access. This assumption is extremely severe and, in fact, pertaining to environmental conditions harsher than those we would find in most practical aerospace applications. It is comparable, for example, to the SEU rates that are caused by solar events in Mercury's orbit [46]. However, such as "stress" test allows us to better unveil how the performance impacts of the rule-based and the D-HMM detection mechanisms set apart from one another in extreme conditions (despite the very short run-time of our benchmark applications).

Using Equation (5.2), we compute the permanent fault probability with a 5-year MTTF to be approximately $1e^{-20}$. This value is used in the low fault rate scenario. To account for the extra wear caused by particle radiation in the high fault rate scenario, we increase the permanent fault rate to $1e^{-5}$ and $1e^{-4}$.

5.8.2 Results

The cache of the system under test has a size of 1024 bytes with 2-way associativity and a 16-byte cache blocks. Due to space limitation, we only present a fraction of the benchmark results in this section. Since the execution times of the Mälardalen benchmarks are extremely short, the high fault rate scenario is the one producing the most noteworthy results. In setting up the SPTA methodology, we select the most frequent memory blocks to be fed to the state space based approach while the remaining blocks are treated with the contention based approach.

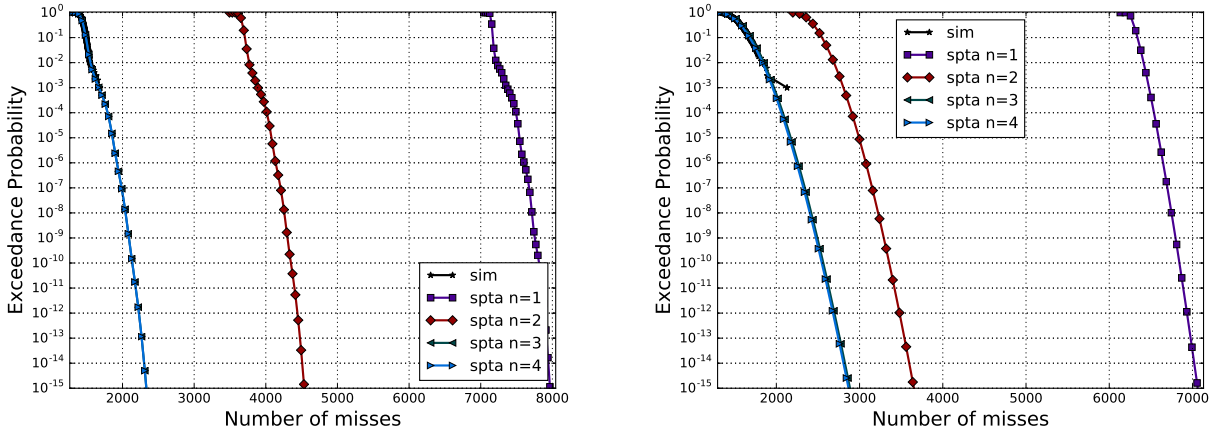
Accuracy of SPTA

We first evaluate the accuracy of the proposed SPTA method when faults occur. Since there are no other established SPTA methods for random caches affected by faults, Figure 5.5 compares the experimental pWCET results with those returned by the proposed SPTA method—using different numbers of blocks n for the state space based approach. The x-axis denotes the number of cache misses, and y-axis indicates the associated exceedance probability, i.e., the probability for a program to encounter more misses than the corresponding number on the x-axis.

The results in Figure 5.5 show that—when a sufficient number of blocks is used in the state space based approach ($n = 3, 4$ for both *fdct* and *compress*)—our SPTA method produces

accurate results. Some minor differences between the simulations and the SPTA result are concentrated around the tails of the distributions. This deviation is due to the relative scarcity of simulation samples, which we limited to constrain our simulations within feasible times. As the number of blocks in the state space based approach decreases, the accuracy of the SPTA method gets compromised and the number of cache misses over-estimated.

We can see that, even though simulations can provide accurate results, one needs $\geq 10^{15}$ simulations when looking at exceedance probabilities of 10^{-15}. This is almost always unfeasible in practice. The number of required simulations could be reduced using MBPTA techniques but is beyond the scope of this research. The proposed SPTA, however, can provide results even for extremely low exceedance probabilties. Its accuracy only depends on the number of blocks used for the state space analysis. Conversely, the calculation time of the SPTA can be reduced at the cost of lower accuracy.



(a) *fdct* benchmark.

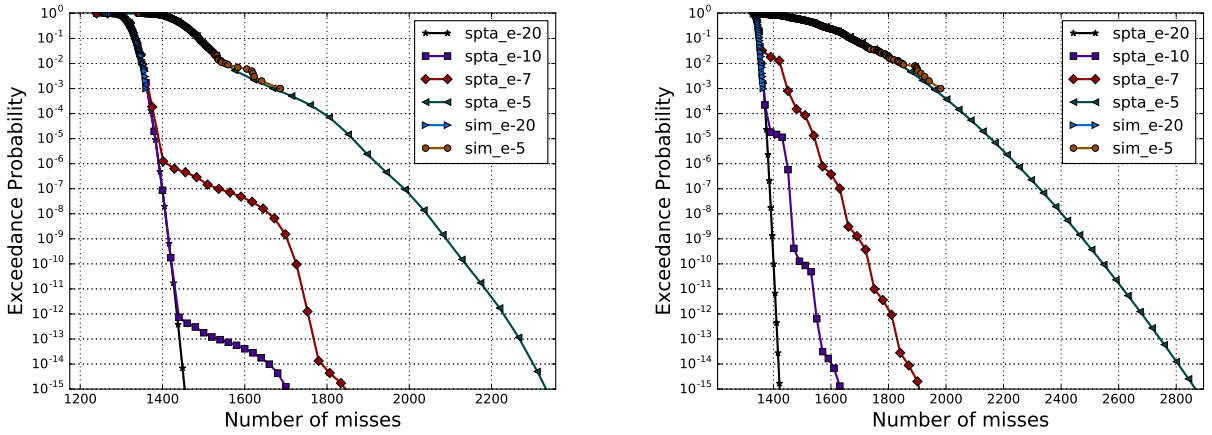
(b) *compress* benchmark.

Figure 5.5 Accuracy of the proposed SPTA with a transient fault probability of $1e^{-10}$ and a permanent fault probability of $1e^{-5}$ per memory access. The number of simulations is set to 1,000. The number of blocks used in the SPTA state space based approach is set to $n = 1, 2, 3, 4$.

The Impact of Faults

Having assessed the performance of the SPTA method per se, we then evaluate the pWCETs of different fault scenarios. The transient fault probability is set to $1e^{-10}$. Since permanent faults can accumulate, they are the ones that most significantly affect the system performance. In our experiments, we evaluates different permanent fault probabilities.

Figure 5.6 presents the pWCETs for different fault scenarios. Our simulations verify that the proposed SPTA provides realistic results. As the permanent fault probability increases, the number of misses increases accordingly, as one would expect. However, we observe that, with time, the number of misses may become dramatically large. For example, in Figure **??**—with permanent fault probability $1e^{-7}$—between the exceedance probability of $1e^{-5}$ and $1e^{-7}$, the number of misses increases significantly. This is due to the fact that some cache sets may be especially sensitive to the accumulated permanent faults. Their pWCET may change significantly and, consequently, affect the final pWCET.



(a) The *fdct* benchmark.

(b) The *compress* benchmark.

Figure 5.6 Study of the impact of faults using the SPTA method. The transient fault probability is set to $1e^{-10}$, and the permanent fault probability to $1e^{-20}, 1e^{-10}, and 1e^{-5}$ per memory access. The simulation is repeated 1,000 times and the number of blocks used in the SPTA state space based approach is set to $n = 4$.

Sensitivity to the Cache Parameters

The cache iteself plays an import role in determining the resulting pWCET distribution. In this section, we use different cache configurations to study their impact on performance and results.

In Figure 5.7, we can see how the cache block size affects the pWCET distributions. As the cache block size increases, the number of cache misses, at low exceedance probabilities, increases. This is because in our model, we assume that, when a permanent fault occurs in a cache block, this cache block will not be re-used in the future. As a result, a permanent fault affects a bigger area of the cache if we are using a larger cache block. This, in turn, entails a larger number of cache misses.

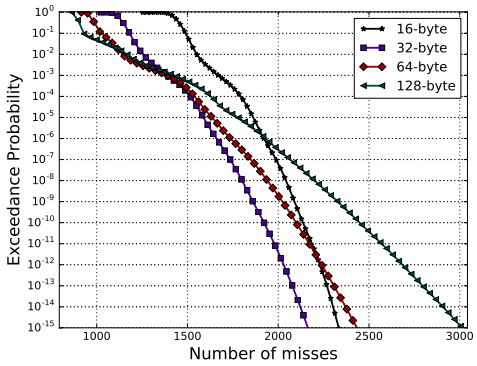


Figure 5.7 Impact of the cache block size on the *fdct* benchmark. The transient and permanent fault probability are $1e^{-10}$, and $1e^{-5}$, respectively. The number of blocks used in the SPTA state space based approach is set to $n = 4$. The cache size is 1024 bytes with 2-way associativity and the cache block size varies from 16-byte to 128-byte.

In Figure 5.8, instead, we increase the cache size to study its impact on the pWCET distributions. Predictably, the pWCET improves as the cache size increases. This is because a larger cache has more cache sets. Thus, the program instructions and data are distributed over a larger number of cache sets, leading to potentially fewer cache conflicts. On the other hand, if the cache size is already sufficiently large for a program's execution, using a larger cache will not necessary improve the pWCET distribution.

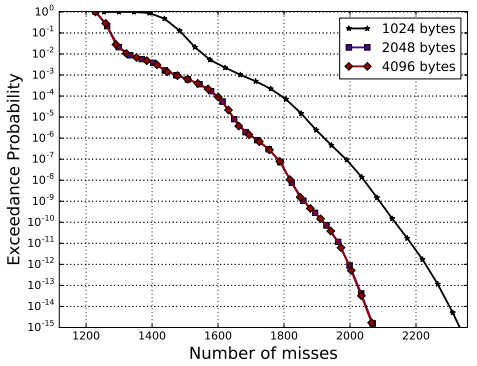


Figure 5.8 Impact of the cache size on *fdct* benchmark. The transient and permanent fault probability are $1e^{-10}$, and $1e^{-5}$, respectively. The number of blocks used in the SPTA state space based approach is set to $n = 4$. The cache block size is 16-byte with 2-way associativity and the cache size varies from 1024 bytes to 4096 bytes.

5.8.3 The Impact of Fault Detection

In the previous sections, we evaluated the pWCETs produced by our SPTA method, assuming that the cache blocks with permanent faults are disabled once they become faulty. However, when both transient and permanent faults occur, it can takes an additional time penalty before being able to classify a fault as permanent. In this section, we evaluate the effects on the pWCETs estimates of different permanent fault detection mechanisms.

For these experiments, we consider the following two fault scenarios:

1. Low fault rate scenario: permanent faults have a probability of $1e^{-20}$ per memory access and transient faults of $1e^{-10}$ per memory access.

2. High fault rate scenario: permanent faults have a probability of $1e^{-4}$ per memory access and transient faults of $1e^{-2}$ per memory

For each scenario, we compare the simulation results obtained using the rule-based and the D-HMM based permanent fault detection mechanisms. We run 1,000 simulations for each one of the detection mechanisms. For the rule-based method, the threshold is set to the empirical value of 4 and the reset period is equal to the execution time of of each benchmark. That is, if 4 faults are detected, the corresponding cache block is considered to be affected by permanent fault and it will be disabled. For D-HMM based detection, the threshold is set to 0.9. If the prediction value exceeds 0.9 (i.e. a 90% belief confidence), the cache block will be disabled. Finally, the SPTA results with perfect fault detection are used as a baseline. Note that we also experimented with other threshold value. As they produced similar results, they are not shown here due to the space limitations.

In the low fault rate scenario, both detection mechanisms result in similar pWCETs, which closely match the results of perfect detection. There is not much difference between pWCETs using different permanent fault detection mechanisms, because the execution times of the benchmarks are short, and with extremely low fault rates, permanent fault rarely occurs, limiting the impact of the detection mechanisms.

In the high fault rate scenario, however, the strengths and weaknesses of different detection mechanisms are no longer masked by the rarity of faults. For any given exceedance probability, the system execution time with rule-based permanent fault detection is much longer than that with D-HMM based permanent fault detection mechanism.

The statistical results of all benchmarks in the high fault rate scenario are displayed in Figure 5.9 as box plots. We observe that for rule-based detection, the execution times are

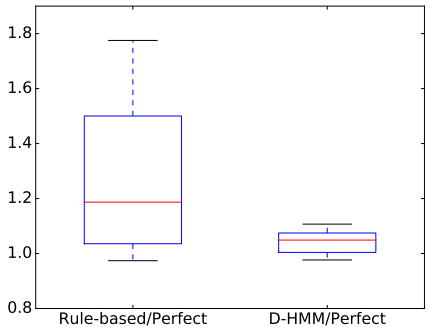


Figure 5.9 Box plots of the latency ratios of "rule-based detection over perfect detection" and "D-HMM based detection over perfect detection" at the exceedance probability of 10^{-3} for all benchmarks in the Mälardalen suite in the high fault rate scenario.

distinctively longer than those obtained using D-HMM detection. The median values are 1.187 and 1.049 respectively.

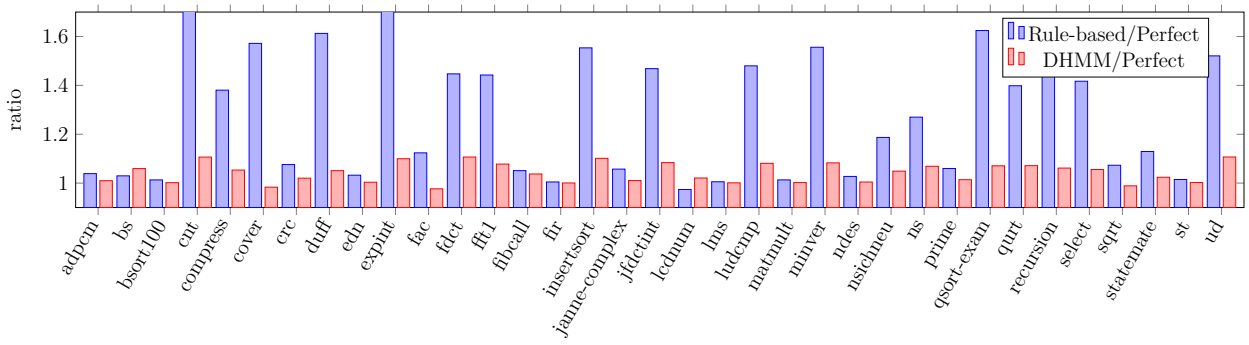


Figure 5.10 The latency ratios in the high fault rate scenario for all benchmarks in the Mälardalen suite.

To better show the effects of detection mechanisms in the high fault rate scenario, the detection-mechanism-to-perfect-detection ratios of all the benchmarks in the Mälardalen suite is presented in Figure 5.10. On the x-axis are the different benchmarks and the y-axis shows the overhead (w.r.t. perfect detection) of rule-based detection execution time to D-HMM based detection execution time for each benchmark—i.e., lower is best. The geometric mean values are 1.256 and 1.042. On average, rule-based detection results a 21% larger overhead than D-HMM based detection.

5.9 Conclusion

In this article, we proposed an SPTA methodology for random caches and single-path programs executing on real-time systems in the presence of faults. The SPTA method consists of two combined approaches: a contention based approach and a state space based approach. With the two approaches, both transient and permanent fault impacts are taken into account, and the lower bound of the number of cache misses is computed. In the SPTA alone, perfect permanent fault detection is assumed. However, this is not the case in a realistic context. Thus, we studied two online permanent fault detection mechanisms: rule-based and D-HMM based detection. We compared the pWCET estimates using both detection mechanisms and found that D-HMM based detection significantly improves the pWCET estimates when compared to rule-based detection. In our future work, we plan to extend our SPTA method to multi-path programs, and study the effect of additional fault tolerant techniques.

CHAPTER 6 ARTICLE 4: STATIC PROBABILISTIC TIMING ANALYSIS WITH A PERMANENT FAULT DETECTION MECHANISM

6.1 Preface

In Chapter 5 [32], we have introduce an SPTA methodology with perfect permananet fault detection. Our experimental findings suggest that, in practice, the specific implementation of a permanent fault detection mechanism is an important factor and may affect pWCET distributions substantially. Hence, it is crucual to take into account the detection technique implementation in the SPTA of random caches. To this end, we propose an SPTA method that integrates permanent error detection, by splitting the SPTA method into two modes to account for the impact of the detection methods, respectively. We investigate one in particular: a rule-based detection technique is added to our SPTA method and its evaluation indicates that this approach is capable of providing safe and tight pWCET estimates.

Authors: Chao Chen, Jacopo Panerati, Imane Hafnaoui and Giovanni Beltrame

Accepted by: 2017 12th IEEE Symposium on Industrial Embedded Systems (SIES)

6.2 Abstract

In recent years, random caches have been proposed as a way to simplify the timing analysis of real-time systems. However, technology-scaling makes caches prone to faults. Fault detection mechanisms can detect permanent faults but they affect the timing analysis of a random cache. This paper introduces a Static Probabilistic Timing Analysis (SPTA) technique that accounts for a permanent fault detection mechanism. The permanent fault detection mechanism periodically checks caches for faults and disables faulty cache blocks to prevent future accesses. The SPTA method operates by periodically switching its run-time between the fault-detection and the no-fault-detection states. This is the first SPTA with a realistic permanent fault detection mechanism. Experiments show that the proposed method always provides safe timing estimations—even when few memory blocks are provided—and accurate results—when sufficient memory blocks are present.

6.3 Introduction

The real-time embedded systems demanded by the aerospace and automotive industries are safety-critical systems that place strict requirements on timing performance. The software

tasks executed by these systems are required to meet their deadlines in all kinds of conditions, including worst case scenarios. Traditionally, designers have used extremely conservative estimations of the execution times extracted from deterministic architectural models. The major shortcoming of this approach is the fact that it can place the Worst Case Execution Time (WCET) extremely far away from the actual maximum time demanded by an application [18]. A rising design trend consists of exploiting probabilistic architectures to obtain better and tighter WCET bounds. Examples of this approaches are already available on commercial processors, such as the ARM cache controller with a pseudo-random replacement strategy[1]. In this paper, a cache with a random replacement policy is employed instead of a traditional, deterministic one such as Least-Recently-Used (LRU). Thus, pathological cases—those that lead to systematic cache misses—are avoided and only low probabilities are associated to extreme cases [90]. These low probabilities ensure that the system execution time matches the safety requirements (e.g. 10^{-9} failures per hour) with tighter WCETs. Probabilistic WCETs (pWCETs) also provide the exceedance probability of a corresponding execution time. In other words, they tell us the probability of encountering a task exceeding a certain execution time to complete its computation.

The vast majority of the research work that focus on WCET estimation techniques for either deterministic or probabilistic architectures is based on the assumption that the hardware is immune to randomly occurring faults. However, performance enhancing techniques—such as technology scaling—have a negative effect on a system's reliability. Transistor shrinking can introduce process variations that increase the components' probability of failure. The use of techniques for power consumption optimization, such as Dynamic Voltage Scaling (DVS), can also increase the failure probability of SRAM cells if the voltage is excessively reduced [117]. The study conducted in [83] paints a morbid picture of what we can expect in the future in terms of failure probabilities due to the scaling of components. The authors report that SRAM cells will be the most affected elements: the probability of failure they predict is 6.1×10^{-13} at $45nm$ and will increase to 2.6×10^{-04} at $12nm$. Another study [58] underlined the issue by reporting that caches in particular will be a major source for performance degradation in future designs. For all of these reasons, we can no longer afford to ignore the effects of fault occurrence when estimating WCET bounds.

On top of permanent faults due to manufacturing processes, we consider dynamic faults from factors such as wear-out effects, etcetera. A non-zero probability of a permanent fault exists at each memory access. The main contribution of this paper is the development of an SPTA method that integrates a permanent fault detection mechanism. To the best of

[1]http://infocenter.arm.com/help/index.jsp

our knowledge, this is the first SPTA with a realistic permanent fault detection mechanism. The SPTA is based on the combined methodology proposed by Altmeyer *et al.* [11]. We extend their approach in our proposed SPTA method by taking faults into account. Since the fault detection mechanism checks for faults periodically, we develop SPTA formulae for two operating modes:

- When the fault detection is off, we develop the SPTA without fault detection, i.e., faulty blocks are regarded as they can still be used.
- When the fault detection is turned on, we develop the SPTA with fault detection which considers the permanent disabling of faulty blocks.

By combining the two modes, our SPTA method can perform timing analysis that integrates a periodic permanent fault detection mechanism.

The organization of the paper is as follows: we examine the related research work in Section 6.4; Section 6.5 presents the assumptions and methods used to describe faults and random caches; our SPTA method is proposed in Section 6.6; we evaluate it using a wide range of benchmarks in Section 6.7; finally Section 7.10 draws the conclusions of the paper.

6.4 Related Work

6.4.1 SPTA

In the literature, three kinds of Probabilistic Timing Analysis (PTA) methods have been proposed: 1. measurement-based [37]; 2. static [118, 27, 39, 11]; and 3. a hybrid of both methods [18]. Caches with random replacement policies are used to make the system probabilistically predictable [90] and PTA computes the pWCET distribution of a program as its timing estimation. Two replacement policies have been proposed: evict-on-access [27] and evict-on-miss replacement. Since the evict-on-miss policy can provide a better pWCET [39], we focus on SPTA with evict-on-miss policy in this paper.

SPTA methods from the early days exploit the reuse distance. Zhou [118] provides a tractable formula for pWCET which, however, is found unsound by Cazorla *et al.* [27] and Altmeyer *et al.* [9]. Quinones *et al.* [90] and Kosmidis *et al.* [66] give other formulae, but these may produce an overestimation of the number of cache hits [40]. Davis *et al.* [39] provide an optimal formula when the only reuse distance is used for calculation.

Altmeyer and Davis [9] propose an SPTA method for single-path programs, which can provide more accurate results compared to previous methods, at the expense of the calculation speed.

Two methods—cache contention and exhaustive state enumeration—are combined to provide a precise and accurate result. Altmeyer *et al.* [11] introduce alternative methods to combine them. Griffin *et al.* [49] propose an SPTA with lossy compression, whose accuracy depends on tunable parameters.

Davis *et al.* [39] propose an SPTA for multi-path programs using path merging. Lesage *et al.* [71] introduce a more accurate SPTA, which is built upon the method in [9] and extended to multi-path programs by calculating the upper bounds of the cache states and reducing the path according to the worst-case execution path expansion.

6.4.2 Faults

The amount of research on PTA in faulty scenarios has recently boomed. Slijepcevic *et al.* [98] use MBPTA to study fault-tolerant real-time systems with random caches. They introduce Degraded Test Mode (DTM) for pWCET calculation, which can compute average and worst-case cases in the presence of faults. More fault scenarios for both transient and permanent faults are introduced in [99], and experiments and simulations show that the impact of faults is negligible for many applications. However, there are some cases where faults affect the system significantly—e.g., when permanent faults decrease the number of cache blocks with severe consequences.

Hardy and Puaut [56] develop an SPTA method for instruction caches with LRU policies. They study permanent faults caused by manufacturing variations. Given the cache miss probability, the pWCET is computed by integrating the fault-free scenario. Hardy *et al.* [55] investigate different reliability mechanisms for permanent faults, and find out that the execution time of programs can be significantly reduced when using simple reliability mechanisms.

Chen *et al.* [31] introduce a state space based SPTA, which can provide an accurate pWCET in the presence of both transient and permanent faults. A perfect permanent fault detection mechanism (which detects permanent faults immediately after their occurrences) is assumed to be adopted. In practice, however, detection cannot be perfect. Therefore, in this paper we develop our SPTA taking this into consideration. To study permanent fault detection effects, Chen *et al.* [30] apply two different fault detection mechanisms and compare their effects, showing that detection mechanism can improve pWCET distributions dramatically. This result is derived from measurements rather than of static analysis.

6.5 System Models

In this section, we present models for a random cache and permanent faults. Their respective impacts on our SPTA methodology are then analyzed.

6.5.1 Random cache model

For our system stochastic, we adopt an N-way set-associative evict-on-miss cache. In this paper, a single-level cache is studied. The cache behavior depends on two policies: placement policy and replacement policy, for these, we adopt modulo placement and random replacement policies, respectively.

A set-associative cache consists of different cache sets in which every cache set is regarded as an N-way fully-associative cache. When a memory block is accessed, it is assigned to a particular set using the modulo placement policy, i.e., several bits of the memory address determine the set. The cache set for each memory block remains the same during a program's execution.

If the memory block is not in the cache, the replacement policy determines the cache behavior for the new memory block. With random replacement, a random cache block in the cache set is evicted to make room for the incoming memory block, thus introducing randomness during a program execution. By doing so, the pathological cases that always evict memory blocks necessary for future accesses on a deterministic architecture can be avoided.

6.5.2 Permanent fault model

Permanent faults are those errors that, when they occur, render a cache block unusable for all future accesses as it will keep producing errors. To enable permanent fault detection, we adopt a periodic scrubbing technique with period T.

In this paper, we are mainly concerned with permanent faults arising from component wear-out effects, rather than those due to the manufacturing process. The aging of its components makes a system more prone to permanent faults as the time goes by. Consequently, there is a permanent fault probability for each memory access. We adopt the model proposed by Panerati *et al.* [86]:

$$f_p(t,T) = \frac{\mathsf{cdf}_{norm}\frac{ln(t)-\mu}{\sigma} - \mathsf{cdf}_{norm}(\frac{ln(t-T)-\mu}{\sigma})}{1-\mathsf{cdf}_{norm}(\frac{ln(t-T)-\mu}{\sigma})}, \tag{6.1}$$

where $f_p(t, T)$ is the probability of a permanent fault for a memory access occurring between time $t - T$ and t and cdf_{norm} is the cumulative density function of the normal distribution.

The mean (μ) and standard deviation (σ) are calculated from a component's Mean Time To Failure (MTTF):

$$\mu = ln(\frac{MTTF^2}{\sqrt{var_{MTTF} + MTTF^2}})$$
$$\sigma = \sqrt{ln(1 + \frac{var_{MTTF}}{MTTF^2})}. \tag{6.2}$$

Equation (6.1) computes the probability of a permanent fault during a specific detection period. In this paper, we use the mechanism proposed in [3] for permanent fault detection. This is an on-line mechanism which detects and disables faulty cache blocks. For each cache block, there is a counter for the number of faults to distinguish permanent faults from transient faults. When the counter value exceeds a threshold, the fault is regarded as permanent and the hardware is reconfigured to disable the corresponding cache block so that the permanent fault is confined. Thus, when a fault is detected, we assume that it is permanent and disable the cache block. In this paper, we only focus on the effects of permanent faults.

6.6 SPTA with Detection

In this section, we review the SPTA method proposed in [11], which consists of two parts: a cache contention method and a state enumeration one. In each part, we show how we extend this SPTA approach to take into consideration faults and their detection mechanism. The SPTA is applied to a fully-associative cache, and it can be extended to a set-associative cache, dealing with each cache set separately.

6.6.1 Cache contention method

Most of the existing SPTA methods use memory traces to perform their analysis. The reuse distance is a metric used extensively in SPTAs. In [11], the reuse distance function rd is formally defined for every access as:

$$rd(e_i, L) = \begin{cases} i - j - 1 & \text{if } \exists e_j \in L : e_j = e_i, \\ & \forall p : j < p < i, e_p \neq e_i \\ \infty & \text{otherwise} \end{cases} \tag{6.3}$$

where $L = [e_1, e_2, ...e_n]$ is a memory trace.

Having defined rd, we can use it as a helper function to perform our SPTA as outlined in the following sections.

The cache contention $con(e_i, L)$ is used to calculate the probability of a cache hit. It depends on previous accesses and provides more precise hit probabilities than other methods, such as the one proposed by Davis *et al.* [39]. Following the methodology proposed in [11], the cache contention is defined as:

$$con(e_i, L) = \begin{cases} \infty & \text{if } rd(e_i, L) = \infty \\ |conS(e_i, L)| & \text{otherwise} \end{cases} \tag{6.4}$$

$$\begin{aligned} conS(e_i, L) = \{e_j \in L | i - rd(e_i, L) < j < i \wedge \hat{P}^N(e_j^{hit}) \neq 0\} \\ \cup \{e_r \in L | r = i - rd(e_i, L)\} \end{aligned} \tag{6.5}$$

We define $\hat{P}^N(e_i^{hit})$ as the lower bound hit probabilities for a cache of associativity N in the scenario in which faults do not occur. Using $con(e_i, L)$ and $rd(e_i, L)$, we can compute $\hat{P}^N(e_i^{hit})$ as follows:

$$\hat{P}^N(e_i^{hit}) = \begin{cases} 0 & con(e_i, L) \geq N \\ (\frac{N-1}{N})^{rd(e_i, L)} & \text{otherwise} \end{cases} \tag{6.6}$$

With the hit probabilities $\hat{P}^N(e_i^{hit})$, we can construct the Probability Mass Function (PMF), which is an essential parameter to estimate the pWCET [11]. In this section, we demonstrate how to calculate hit probabilities using the cache contention method (refer to [11] for additional detail on the use of PMFs). This is the starting point from which we build our method to include faults and their detection mechanism.

6.6.2 Extension of the cache contention method

Due to permanent faults, some cache blocks may be unusable. For this reason, we analyze $N + 1$ different hit probabilities, representing different numbers of faulty cache blocks. We use $\hat{P}_F^k(e_i^{hit})$ $(k \in \mathbb{N}, 0 \leq k \leq N)$ to denote the k-th lower bound probability, i.e., with k available blocks ($N - k$ blocks are detected as permanently faulty). Note that for $k = 0$, no cache blocks are available, which makes the cache miss probability to 1. For these groups, we skip the calculation of the hit probability.

The permanent fault detection mechanism detects faults periodically. Therefore, it has two operating modes: the time when fault detection is actively performed and the time it spends in an idle state. We identify them as **Active Mode** and **Idle Mode**, respectively, in the following sections.

We use % to indicate the modulo operator. It is used to find remainder after division of two numbers and it is defined as

$$i\%j = i - j \cdot \lfloor i/j \rfloor$$

Let T_a be the period in terms of number of memory accesses. For a memory block e_i, if $i\%T_a = 0$, the fault detection mechanism is in the **Active Mode**. Otherwise, it is in **Idle Mode**.

In the **Active Mode**, the fault detection mechanism is able to detect the permanent faults and the following equation can be extracted for the k-th group probability:

$$\hat{P}_F^k(e_i^{hit}) = \prod_{j=m}^{i} a_p^j \cdot b_k^i \cdot \hat{P}^k(e_i^{hit}) \tag{6.7}$$

where $\hat{P}^k(e_i^{hit})$ is computed using Equation (6.6). a_p is a parameter relating to permanent faults such that a_p^i computes the probability of not having a permanent fault in a memory block e_i. Given that n_a^i is the number of memory accesses between the accesses of e_{i-1} and e_i ($n_a^i = 1$ for a fully-associative cache), and f_p is the probability of the occurrence of a permanent fault for each memory access, then we have:

$$a_p^i = (1 - f_p)^{n_a^i} \tag{6.8}$$

m is the memory index such that $m < i \wedge e_{m-1} = e_i \wedge \forall m < p < i : e_p \neq e_i$, i.e., it accounts for all subsequent memory blocks after the previous access of the same memory block e_i. As faults are independent from one another, the product term $\prod_{j=m}^{i} a_p^j$ denotes possible evictions due to faults.

To take into account each group, we define d_k^i as the occurrence probability of the k-th set of probabilities before accessing e_i. For $k = N$, we have $d_k^i \geq d_k^j,\ i \leq j$ since, as time progresses, the probability of a system without faulty blocks decreases. For $k = 0$, we have $d_k^i \leq d_k^j,\ i \leq j$—since faulty blocks cannot be accessed any longer—thus, the probability without the available blocks increases as time goes by. When $0 < k < N$, d_k^i may increase or

decrease depending on the access time. We use the following equation to calculate it:

$$d_k^i = \sum_{j=k}^{N} \begin{pmatrix} j \\ k \end{pmatrix} (1 - a_p^i)^{j-k} (a_p^i)^k d_j^{i-1} \tag{6.9}$$

where

$$d_k^0 = \begin{cases} 1 & \text{if } \ k = N \\ 0 & \text{otherwise} \end{cases} \tag{6.10}$$

When a memory block is accessed, we need to add it to cache. However, the occurrence probability of each group is different. We define a parameter b_k^i to denote the k-th group occurrence probability for the last access of the same memory block and it represents the probability of putting the memory block into cache. We can calculate b_k^i as follows

$$b_k^i = \begin{cases} d_k^j & \text{if } \exists e_j \in L : e_j = e_i, \\ & \forall p : j < p < i, e_p \neq e_i \\ 1 & \text{otherwise} \end{cases} \tag{6.11}$$

With $\prod_{j=m}^{i} a_p^j$ and b_k^i, we can calculate $\hat{P}^N(e_i^{hit})$ using Equation (6.7).

In **Idle Mode**, we consider the fault detection mechanism to be turned off. In this case,

$$\hat{P}_F^k(e_i^{hit}) = \prod_{j=m}^{i} a_p^j \cdot min(b_k^i, b_k^{i-i\%T_m}) \cdot \hat{P}^k(e_i^{hit}) \tag{6.12}$$

where i is the index of the current memory block, and $i - i\%T_m$ is the memory block for which a previous fault detection occurs. b_k^i is calculated using Equation (6.11). Because the fault detection is not active, when another fault occurs, it cannot be detected. We use $min(b_k^i, b_k^{i-i\%T_m})$—the minimal value between b_k^i and $b_k^{i-i\%T_m}$—to calculate the occurrence probability of its k-th group, since it computes a pessimistic occurrence probability and thus provides a lower bound hit probability.

Equations (6.7) and (6.12) calculate the estimated hit probabilities with and without permanent fault detection mechanism, respectively. We exploit both equations and switch between them periodically to account for the effects of fault detection in this SPTA cache contention-based method. The hit probability with fault detection $\hat{P}_F(e_i^{hit})$ can be computed by summing the probabilities of all groups as:

$$\hat{P}_F(e_i^{hit}) = \sum_{j=1}^{N} \hat{P}_F^j(e_i^{hit}) \tag{6.13}$$

6.6.3 State enumeration method

In addition to the contention-based method, Altmeyer *et al.* [11] propose a state enumeration-based method. This method takes into account all states to obtain more precise results. A set of memory blocks is defined as $E \subseteq \mathbb{E}$, $P \in \mathbb{R}$ denotes the probability and $D : \mathbb{N} \to \mathbb{R}$ represents the cache miss distribution. Thus, a cache state is defined by $CS = (E, P, D)$.

State enumeration provides accurate results at the expense of the computation time that increases exponentially because of the large number of states. To avoid state explosion, a set of memory blocks R is selected, while the remaining blocks are treated using the contention-based method. The symbol $\perp$ denotes an empty cache block or a memory block $e \notin R$. The number of selected memory blocks is denoted by n, i.e. $n = |R|$, and we put the n most frequently used memory blocks into the set R. For a memory block e, if $e \in R$, we treat it with the state enumeration method. Otherwise, we apply the cache contention method. The state enumeration method and the cache contention method produce two independent pWCETs, and the final pWCET result obtained by their convolution.

An initial cache state is denoted as

$$CS_{init} = (\{\perp, ... \perp\}, 1, D)$$

with

$$D(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.14}$$

Every time a memory block e is accessed, the state space changes and an update function u describes how to quantify this event:

$$u((E, P, D), e) = \begin{cases} \{(E, P, D)\} & \text{if } e \in R \wedge e \in E \\ miss((E, P, D), e) & \text{otherwise} \end{cases} \tag{6.15}$$

with

$$miss((E, P, D), e) = \{((E \setminus e') \cup \{e\}, P \cdot \frac{1}{N}, D') | e' \in E\} \tag{6.16}$$

and

$$D'(x) = \begin{cases} D(x) & \text{if } e \notin R \\ 0 & \text{if } e \in R \wedge x = 0 \\ D(x-1) & \text{otherwise} \end{cases} \tag{6.17}$$

We also define a merge operation $\uplus$, that can be used to reduce the number of states that share the same memory blocks:

$$(E_1, P_1, D_1) \uplus (E_2, P_2, D_2) = \begin{cases} \{(E_1, P_1 + P_2, (\frac{P_1}{P_1+P_2} D_1) \oplus (\frac{P_2}{P_1+P_2} D_2))\} & E_1 = E_2 \\ \{(E_1, P_1, D_1), (E_2, P_2, D_2)\} & \text{otherwise} \end{cases} \tag{6.18}$$

where $\oplus$ is the summation of the distributions and $p \cdot D$ is the element-wise multiplication of D by p.

Using the initial cache state CS_{init} and the update function u, we can calculate how the cache state changes as well as the cache miss distribution. The final cache miss distribution is computed as the sum of all the cache miss distributions, weighted by the corresponding occurrence probability P.

6.6.4 Extension of the state enumeration method

To deal with permanent fault effects, we apply a state space-based method. We introduced a modification to incorporate the state enumeration method, which causes additional state changes at every memory access to account for the presence of faults.

Similarly to what we did for the contention-based method, we have to deal with two different operating modes: **Active Mode** and **Idle Mode**. In either mode, we analyze both the fault occurrence and the effects of fault detection.

Before the update function u, a permanent fault function pf is added to account for additional state changes caused by faults. A fault detection function fd is used to represent different detection effects. The $miss$ function in Equation (6.16) must also be modified accordingly when in the presence of faults.

To take into account the fault detection mechanism, we extend the 3-tuple cache state $CS = (E, P, D)$ to a 4-tuple cache state $CS = (E, P, D, A)$, by introducing an additional element $A \in \mathbb{N}$. A stores the k-th group index k from which new states come in the pf function and its initial value is $A = -1$. If $A \neq -1$, then a fault has occurred in the current state but this has not been detected yet. The merge operation (7.8) can be used when both E and A are equal for two cache states.

In **Active Mode**, we deal with the case in which detection is present. We define S_k as a cache state set which contains $k \in \mathbb{N}, 0 \leq k \leq N$ cache blocks without permanent faults, i.e., it denotes the k-th group. Let $CS = (E, P, D, A) \in S_i$ be a cache state before permanent

faults, and $CS' = (E', P', D, A') \in S_j (j \leq i)$ be a cache state after a permanent fault occurs. Note that faults do not change D (and $j \leq i$) because a state can stay in S_i without any fault, or it moves to S_j with fewer available blocks because of the induced faults. The permanent fault effect is calculated as

$$pf((E, P, D, A)) = \{(E', P \cdot (a_p^i)^{|E'|} \cdot (1 - a_p^i)^{|E \setminus E'|}, D, A') \\ |E' \subseteq E\} \tag{6.19}$$

where a_p^i is the probability without a permanent fault as it was calculated in Equation (6.8) and A' is

$$A' = \begin{cases} i & \text{if } j < i \wedge A = -1 \\ A & \text{otherwise} \end{cases} \tag{6.20}$$

We can see that, due to permanent faults, some faults may cause permanent damage on cache blocks, therefore changing the state from S_i to S_j. Furthermore, A is used to check if a permanent fault has been detected or not.

The fault detection function fd describes the detection mechanism. In this paper, we reset A to its original value -1 to indicate that faults are detected, so we have

$$fd((E, P, D, A)) = (E, P, D, -1) \tag{6.21}$$

The *miss* function defines how the system behaves in the case of a cache miss. For $CS = (E, P, D, A) \in S_i$, we have:

$$miss((E, P, D, A), e) = \{((E \setminus e') \cup \{e\}, P \cdot \frac{1}{i}, D', A) | e' \in E\} \tag{6.22}$$

This is similar to the *miss* function without faults. However, for a different cache set S_i, we have a different number of available cache blocks. Thus, we need to use i for the probability calculation, instead of the cache associativity N.

In **Idle Mode**, the permanent fault function pf is the same as that in Active Mode and there is no fd function. On the contrary, the *miss* function is different. This is due to the fact that some cache blocks may have permanent faults and yet they may not be detected in this case, which makes the system unaware of some faulty cache blocks and unable to disable them. As a result, some faulty blocks may be used to store incoming memory blocks.

With $CS = (E, P, D, A) \in S_i$, the $miss$ function is computed as

$$\begin{aligned} miss((E,P,D,A),e) = \{((E \setminus e') \cup \{e\}, P \cdot \frac{1}{i}, D', A) | e' \in E \\ \wedge A = -1\} \cup \{((E \setminus e') \cup \{e\}, P \cdot \frac{1}{A}, D', A) | e' \in E \wedge A \neq -1\} \\ \cup \{(E, P \cdot \frac{A-i}{A}, D', A) | A \neq -1\} \end{aligned} \tag{6.23}$$

This $miss$ function consists of three parts. In the first part, we consider the states whose faults have been detected. In the second part, we cope with the states whose faults have not been detected, and the cache blocks without faults are used to store data. In the third part, faults have not been detected, and the faulty cache blocks are selected, resulting in no new states. The union of the three parts is the state set after a cache miss.

With periodic switches between **Active Mode** and **Idle Mode**, we are able to analyze the system considering the effects of permanent fault detection.

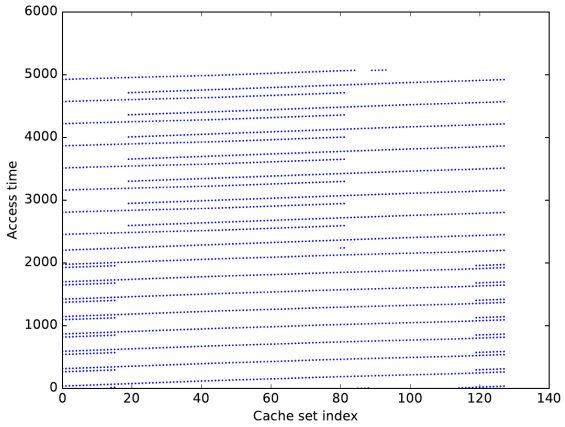
6.7 Experimental Evaluation

We evaluate the accuracy of the proposed SPTA methodology in this section. With our method, we investigate different scenarios and study the fault and detection effects on timing analysis.

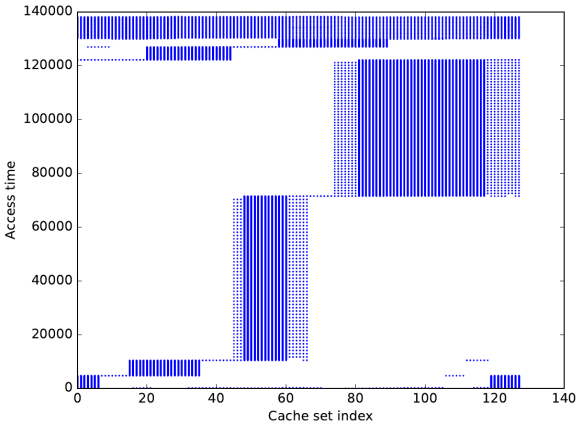
6.7.1 Experimental Setup

To evaluate the proposed approach, we select the Mälardalen benchmark suite [52]—which is used frequently for WCET estimations, including pWCET analysis (e.g., in [11])—to perform our experiments. We assume that the system is equipped with an L1 cache and a main memory. We use only instruction caches for evaluations, but our approach can be applied to data caches as well.

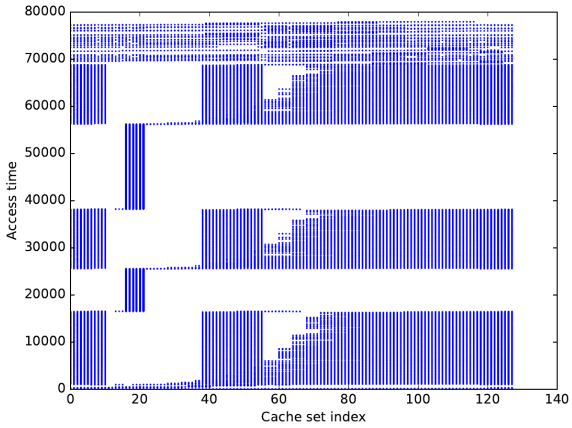
To obtain the memory traces for the input, we generate binary executable files for the benchmarks and use a instruction set simulator, `gem5` [23], to generate traces. We use the default ARM processor in the `gem5` for simulation, and our experiments are performed on a Linux system. In the compilation of the executables, we choose to use statically linked libraries and assume that a Floating Point Unit exists for floating point operations. We generate benchmark traces using the default inputs.



(a) *fdct* memory trace access pattern.



(b) *edn* memory trace access pattern.

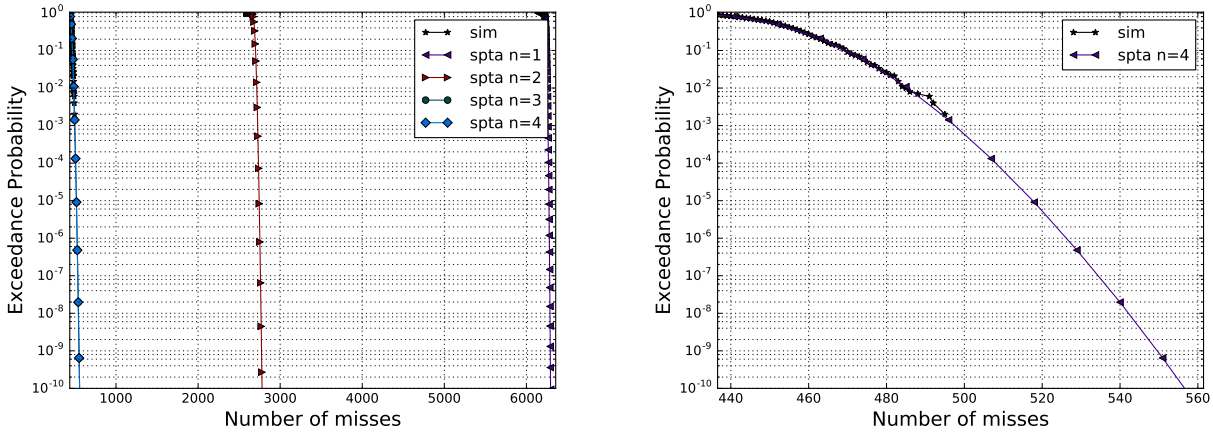


(c) *adpcm* memory trace access pattern.

Figure 6.1 Memory trace access patterns for selected benchmarks. Each memory block access is plotted as a dot. The x-axis represents the cache set index (0-127) in which the memory block is stored, and the y-axis shows the access time of the memory block.

6.7.2 Benchmark Selection

Due to space limitations, we have selected 3 benchmarks for evaluation: *fdct*, *edn* and *adpcm*. Figure 6.1 shows the memory trace access patterns of the benchmarks, from which we observe which cache sets are accessed with respect to time. We assume that each access consumes the same amount of time, which is not necessarily always true. For this reason, Figure 6.1 is an approximate access pattern. We can see that the accesses for *fdct* are sequential, i.e., the system fetches memory blocks in consecutive cache set indexes continually. For *edn*, a number of adjacent cache sets are accessed continuously over a period of time, and then a



(a) *fdct* benchmark with different numbers of memory blocks.

(b) Zoomed in *fdct* benchmark accuracy verification.

Figure 6.2 SPTA accuracy estimation for *fdct*. The number of blocks used in the SPTA state space based approach is set to $n = 1, 2, 3, 4$.

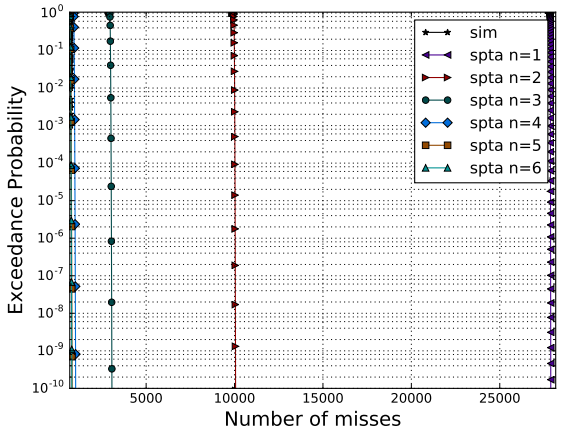
different batch of cache sets is repeatedly accessed in the following period. For *adpcm*, we observe that the cache sets are repeatedly accessed with a specific access pattern.

6.7.3 Accuracy Verification

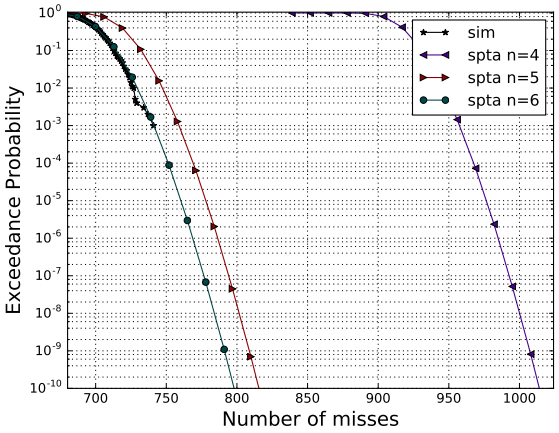
Our proposed approach is based on a combined method using a cache contention method and a state enumeration method, which results in a compromise between calculation time and accuracy. The accuracy of the result improves as the number of memory blocks $n = 1, 2, 3...$ applied to state enumeration method increases.

We use a cache with a size of 1024 bytes, 2-way associativity and 16-byte cache block for evaluation. To verify the accuracy, we use the results obtained from $1,000$ simulations as the baseline. The permanent fault probability per memory access is set to $f_p = 1e{-}20$, which is the value for a 1MHz processor with 5-year MTTF using Equation (6.1). We set the fault detection period to $T_a = 100$ memory accesses in all experiments.

Figures 6.3 – 6.4 show the results obtained for the benchmarks previously discussed. The results are expressed as probabilistic WCET (pWCET), i.e., the exceedance probabilities with respect to corresponding execution times. The x-axis denotes number of cache misses. The greater the number of cache misses, the longer the execution time of the program runs. The y-axis represents the exceedance probability for a cache miss number and the lowest exceedance probability is set to 10^{-10}.

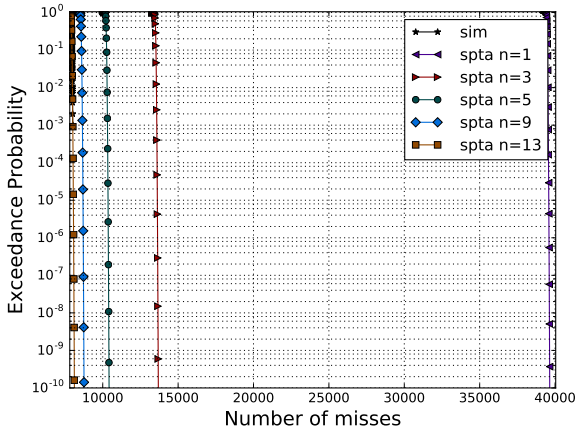


(a) *edn* benchmark with different numbers of memory blocks.

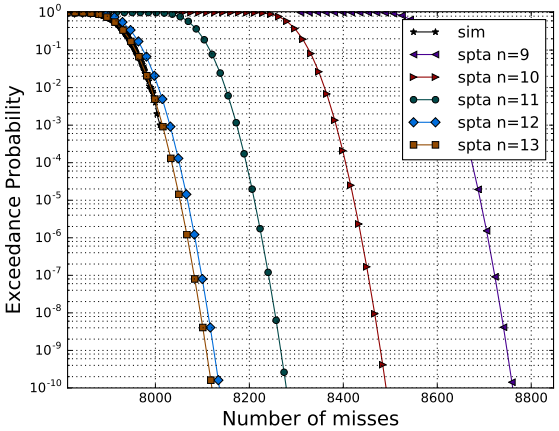


(b) Zoomed in *edn* benchmark accuracy verification.

Figure 6.3 SPTA accuracy estimation for *edn*. The number of blocks used in the SPTA state space based approach is set to $n = 1, 2, 3, 4, 5, 6$.



(a) *adpcm* benchmark with different numbers of memory blocks.



(b) Zoomed in *adpcm* benchmark accuracy verification.

Figure 6.4 SPTA accuracy estimation for *adpcm*. The number of blocks used in the SPTA state space based approach is set to $n = 1, 3, 5, 9, 10, 11, 12, 13$.

The results from our method can be compared with the results obtained from simulations. Let m_n be the number of cache misses with n memory blocks for the state enumeration method, and m_b be the baseline cache miss number at the exceedance probability of 10^{-10}. To obtain m_b, we use the SPTA result which matches the simulation result at higher exceedance probabilities, i.e., $m_b = m_4, m_6, m_{13}$ for *fdct*, *edn* and *adpcm*, respectively. We define the

ratio ζ as

$$\zeta = \frac{m_n}{m_b} \times 100\%$$

This compares the SPTA result with the memory block number n with the base simulation result, and indicates the precision of our SPTA method.

For $n = 1$, the results from the SPTA are much more pessimistic than those from simulations. For *fdct*, *edn* and *adpcm*, the ratios ζ are about 1100%, 3370% and 480% more pessimistic, respectively. For $n = 2$, ζ's are 520%, 1250% and 170%. We can see that when few memory blocks are used, our method produces pessimistic and hence safe results. However, as the number of memory blocks increases, the accuracy of the estimation improves accordingly.

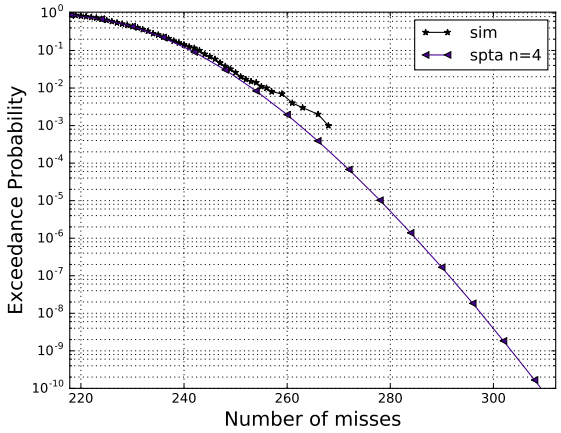
As more memory blocks are applied, the results from our approach are the same as those from simulations. However, for different benchmarks, the number of memory blocks required to reach an acceptable accuracy may differ. For *fdct* (Figure 6.3), for example, we can reach relatively accurate results with $n = 3$. When n is further increased, the SPTA results are very close and might even be matching. We can see that the pWCET curve with $n = 3$ overlaps the pWCET curve with $n = 4$ in Figure 6.2(a). On the other hand, the requirement to reach the same level of accuracy for *edn* and *adpcm* is $n = 6$ (Figure 6.3(b)) and $n = 13$ (Figure 6.4(b)) respectively, since their code sizes are larger, and hence utilizes more memory blocks. When the pWCETs from SPTA match those from simulations, we claim that sufficient memory blocks are present in the state enumeration method. For *fdct*, *edn* and *adpcm*, the sufficient number of memory blocks are $n = 4, 6, 13$, respectively.

6.7.4 Cache Performance

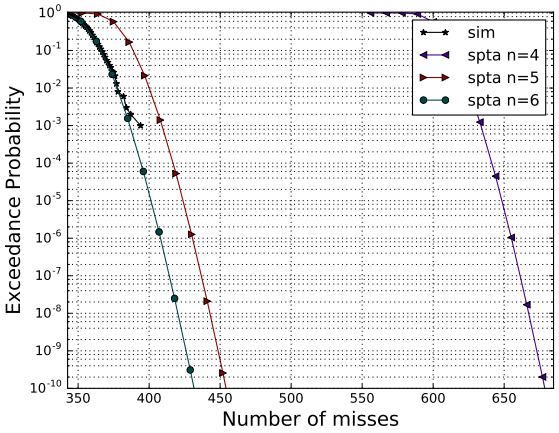
In this section, we change the cache configurations to study the cache impacts on system performance and further verify the accuracy of our method.

We change the cache block size from 16 to 32 bytes, while the rest of the configuration remains the same as described in Section 6.7.2. Figure 6.5 illustrates the pWCETs of the same benchmarks. The figures show that when we double the cache block size, our method pWCETs still match simulation pWCETs. In addition, when larger cache blocks are used, cache misses are significantly reduced, i.e., m_n decreases dramatically in Figure 6.5 when compared to the results in Figures 6.3, **??** and 6.4. This is due to the fact that, when a cache miss happens, memory blocks are fetched and stored in cache for future accesses. Compared to a smaller cache block size which is forced to evict certain blocks, a larger cache block size effectively increases the number of cache hits and reduces cache misses.

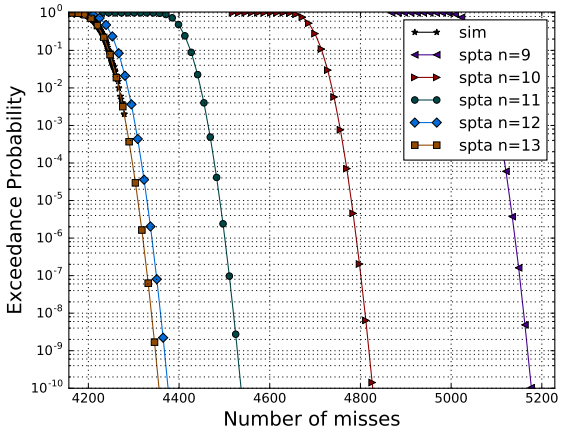
Next, we set the cache to a 4-way associativity cache, instead of a 2-way. The number



(a) *fdct* with the number of memory blocks $n = 4$.



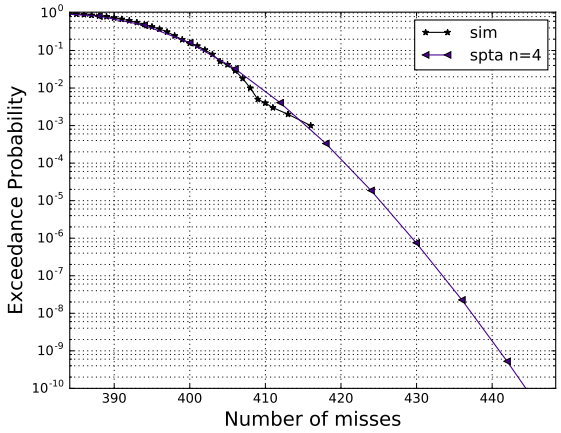
(b) *edn* with the number of memory blocks $n = 4, 5, 6$.



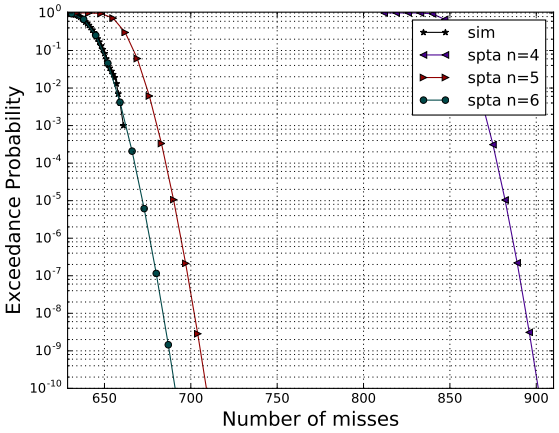
(c) *adpcm* with the number of memory blocks $n = 9, 10, 11, 12, 13$.

Figure 6.5 SPTA accuracy estimation with 32-byte cache blocks.

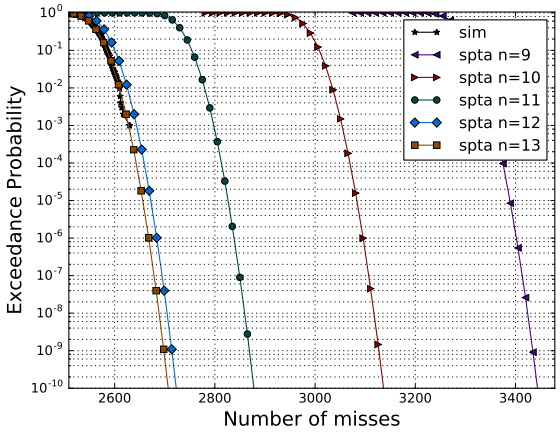
of cache sets remains the same, which increases the cache size to 2048 bytes. The other configuration parameters are kept the same as those in Section 6.7.2. Figure 6.6 displays the pWCETs of the benchmarks and it shows that accurate results can still be computed with a sufficient number of memory blocks. The pWCETs using 4-way caches and their variations with different memory blocks are similar to those using 2-way caches, except that the number of cache misses is different. It shows that for *fdct* and *edn*, the number of cache misses m_n grows when using 4-way cache with reduced a cache block size, but m_n decreases for *adpcm*. Therefore there is no consistent conclusion on whether 2-way caches with a larger cache block size perform better.



(a) *fdct* with the number of memory blocks $n = 4$.



(b) *edn* with the number of memory blocks $n = 4, 5, 6$.



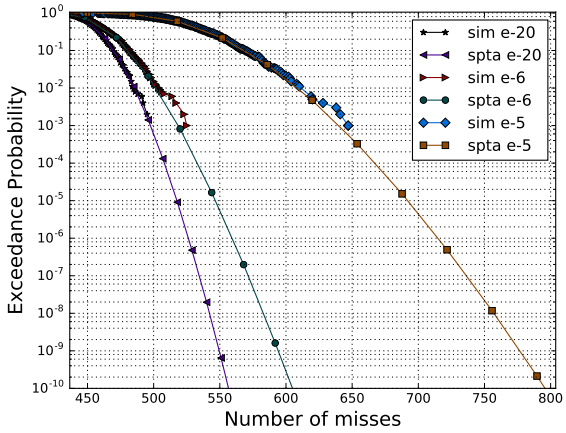
(c) *adpcm* with the number of memory blocks $n = 9, 10, 11, 12, 13$.

Figure 6.6 SPTA accuracy estimation with 4-way cache and the same number of cache sets.

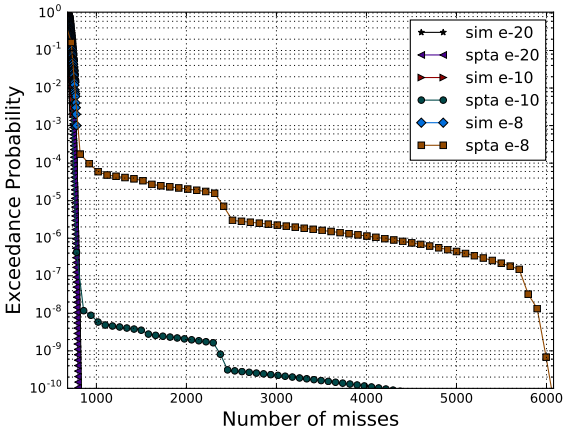
6.7.5 Fault and Detection Effects

From Equation (6.1), we can see that fault probability varies with time. In this section, we study fault and their detection effects in different fault scenarios. The cache configuration is the same as in Section 6.7.2. However, we increase the fault probabilities gradually to investigate their impacts in extreme conditions.

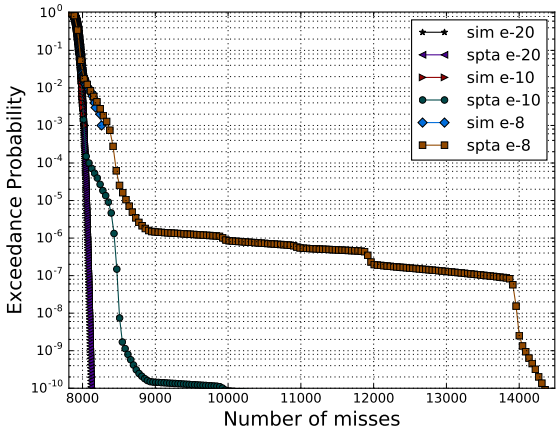
Figure 6.7 compares the simulation pWCETs and SPTA pWCETs, which shows that our SPTA method can provide accurate results in different permanent fault scenarios. We apply different permanent fault probabilities to the benchmarks to see how faults affect system



(a) *fdct* with $n = 4$ and fault probability e-20, e-6, e-5.



(b) *edn* with $n = 6$ and fault probability e-20, e-10, e-8.



(c) *adpcm* with $n = 13$ and fault probability e-20, e-10, e-8.

Figure 6.7 SPTA accuracy estimation in different fault scenarios for the benchmarks.

timing behaviors. We assume that cache blocks have the same property and they degrade in the same way. Thus, at each cache access, each block has the specified permanent fault occurrence probability. In Figure 6.7(a), we select fault probabilities e−20, e−6 and e−5 respectively to plot pWCETs with significant differences. In Figures 6.7(b) and 6.7(c), the fault probability is limited to up to e−8, since as the fault probability increases further to e−5, the number of cache misses becomes much larger at the exceedance probability of 10^{-10}, and cannot be properly displayed in the figure.

Intuitively, as more faults occur in the system, the number of cache miss increases. For large benchmarks (Figures 6.7(b) and 6.7(c)), the system is more affected by the faults, i.e., the

cache miss number increases considerably at the exceedance probability of 10^{-10} compared to the number of cache misses with fault probability e−20. Due to different characteristics of the benchmarks, their timing estimations vary differently as well. In Figure 6.7(a), the number of cache misses increases gradually. In Figures 6.7(b) and 6.7(c), we notice dramatic increase in the number of cache misses . We observe that in Figure 6.7(b) with probability of e−8, when the exceedance probability decreases from 10^{-6} to 10^{-7}, the cache miss number doubles from 2500 to 5000. Such a phenomenon appears in Figure 6.7(c) as well, but does not appear in Figure 6.7(a).

6.8 Conclusion

Random caches are becoming increasingly important but they are increasingly prone to faults because of various reasons, such technology scaling and aging effects. In this paper, we proposed an SPTA methodology accounting for the presence of permanent faults and a periodic permanent fault detection mechanism. Our methodology is based on a combined state-of-the-art approach for random caches for single-path programs. The combined approach consists of two methods: a cache contention method and a state enumeration method. We analyzed the fault occurrence and the effects of fault detection for each method separately. Due to periodic fault detection, we developed our SPTA methods for an **Active Mode** and an **Idle Mode** (i.e. with and without fault detection) and the system switches periodically between the two. Experimental results showed that our approach can provide safe timing estimations using few memory blocks, and its accuracy improves as the number of used memory blocks increases. We verified the accuracy of our proposed method using different cache configurations and fault probabilities, and studied both the impact of the cache and faults on timing behaviors. In our future work, we intend to further improve the accuracy of the approach when using only few memory blocks, and investigate faults and their detection mechanisms in multi-path programs.

CHAPTER 7 ARTICLE 5: AN ADAPTIVE MARKOV MODEL FOR THE TIMING ANALYSIS OF PROBABILISTIC CACHES

7.1 Preface

In Chapter 3 [31], we have proposed a Markov chain based SPTA method. In this article, we extend that work as follows: we formally present a general Markov chain framework; we use examples to show in detail how the proposed method works for time-randomized caches; we demonstrate the safety of the pWCET adaptively obtained from the Markov model using limited number of memory blocks; we extende the Markov chain approach to write-back caches with a write-allocate policy; we implement a state-of-the-art SPTA methodology and compare this method and our proposed method; we compare the timing behaviors of real-time systems with time-randomized caches and Least Recently Used (LRU) caches, respectively. The results allow us to support the claims of previous studies.

Authors: Chao Chen and Giovanni Beltrame

Accepted by: ACM Transactions on Design Automation of Electronic Systems (TODAES) [33].

7.2 Abstract

Accurate timing prediction for real-time embedded software execution is becoming a problem due to the increasing complexity of computer architecture, and the presence of mixed-criticality workloads. Probabilistic caches were proposed to set bounds to Worst Case Execution Time (WCET) estimates and help designers improve real-time embedded system resource use. Static Probabilistic Timing Analysis (SPTA) for probabilistic caches is nevertheless difficult to perform, because cache accesses depend on execution history, and the computational complexity of SPTA makes it intractable for calculation as the number of accesses increases. In this paper, we explore and improve SPTA for caches with evict-on-miss random replacement policy using a state space modeling technique. A non-homogeneous Markov model is employed for single-path programs in discrete-time finite state space representation. To make this Markov model tractable, we limit the number of states and use an adaptive method for state modification. Experiments show that compared to the state-of-the-art methodology, the proposed adaptive Markov chain approach provides better results at the occurrence probability of 10^{-15}: in terms of accuracy, the state-of-the-art SPTA results are more conservative, by 11% more on average. In terms of computation time, our approach

is not significantly different from the state-of-the-art SPTA.

7.3 Introduction

A time-critical embedded computing system, such as a satellite on-board computer, requires accurate timing prediction of software execution. If events are not managed within a certain timeframe, the result may be catastrophic. Historically, these systems were kept at a minimum of complexity to minimize the occurrence of failures and to maintain high timing predictability. However, to address the increasing complexity of applications and their corresponding need for performance, more advanced architectures using multi-stage pipelines, several memory hierarchy levels and even Multi-Processor System-on-Chip (MPSoC) designs [77] are proposed.

These traditional deterministic computer architectures make software timing behavior almost impossible to accurately predict. Normally, the execution time of an application on a deterministic architecture follows a distribution that might have some corner cases which are beyond normal operation. A conservative estimation will place the Worst Case Execution Time (WCET) far away from the actual maximum time used by the application [18], especially when considering possible interactions with other tasks. This would lead to a large overestimation of the computing resources needed for the task [27].

To help predict timing behavior, probabilistic real-time systems were introduced. Such systems have very low pathological occurrence probabilities, which are hard to test and predict. Quinones *et al.* [90] study an instruction cache with randomized replacement (random replacement pre-existed their work), showing that it provides tighter bounds for pathological cases in which systematic cache misses happen and their probabilistic WCET (pWCET) can be empirically derived by experiments. Moreover, some commercial real-time systems have adopted time-randomized caches as well, such as the ARM processor with a pseudo-random cache replacement policy[1].

Two timing analysis techniques are proposed in literature [116]: measurement based timing analysis and static timing analysis, which have their probabilistic counterparts in Measurement Based Probabilistic Timing Analysis (MBPTA) and Static Probabilistic Timing Analysis (SPTA) respectively. The result of SPTA and MBPTA is expressed in terms of pWCET, i.e. an exceedance function that shows the probability of an application to exceed a given execution time.

MBPTA is an empirical method: it is based on repeated testing of an application to estimate

[1] `http://infocenter.arm.com/help/index.jsp/`

its timing probability distribution. Generally, MBPTA requires a large amount of data from simulations or testing on real systems to get accurate results. Cucu-Grosjean *et al.* [37] propose an MBPTA methodology based on Extreme Value Theory (EVT) [41, 13], which needs only a few hundred runs for MBPTA.

SPTA uses a different approach: it is based on detailed knowledge of software and hardware. Together with simulation models and theoretical analysis, a precise timing analysis or a timing bound can be obtained. For caches, several variables are used for the bound calculation, e.g. *reuse distance* and *cache associativity*. *Reuse distance* defines the degree of separation between two accesses to the same memory address. pWCET estimates can be computed with the help of *reuse distance* and *cache associativity*.

In this paper, we present a methodology for SPTA for set-associative instruction and data caches with random replacement policy, which is based on the Markov chain model in [31]. It takes single-path programs as inputs and computes exceedance probabilities with respect to execution time (the number of processor cycles in our simulations). The calculation is performed using state space techniques, and it is based on a non-homogeneous Markov chain model [97]. At every step, the current status of the system can be represented as a state vector with a corresponding probability vector, and the transition matrix for next step is calculated accordingly. To perform timing analysis, timing distribution vectors–which are used for timing representation and analysis–are assigned for each state. We implement another precise SPTA methodology [11] that can also be used for accurate timing analysis, and find out that it provides WCET bounds that are 11% looser on average in terms of geometric mean than the results from the proposed method, while having a similar computational cost. With the proposed Markov chain based method, we can evaluate cache impacts on system performance, which helps the design of real-time embedded systems.

The rest of the paper is organized as follows: related work is discussed in Section 7.4; system modeling is explained in Section 7.5; timing analysis for the system is demonstrated in Section 7.6; an adaptive method is introduced in Section 7.7 to limit the computational complexity of the Markov chain based model; real-world benchmarks are evaluated in Section 7.9; and finally Section 7.10 draws come concluding remarks.

7.4 Related Work

There have been few research efforts on timing analysis for probabilistic systems. Bernat *et al.* [18] develop a WCET analysis method for probabilistic hard real-time systems, in which the concept of a probabilistic system–whose execution deadline must be met by given

probabilities–is introduced. They use the notion of execution profiles for timing representation and analysis. To help determine the WCET of programs, Bernat *et al.* [21] propose an approach based on *copulas*. It uses the dependence structure description of programs for computing the WCET, and when unavailable, it provides a lower bound.

Traditional computing systems are deterministic, and their timing analysis depend on execution history, whose computational complexity increases exponentially as the program executes. To reduce the dependency on execution history, probabilistic systems are introduced, implemented in hardware and software.

By using hardware techniques, programmers do not need to modify software and the system WCET can be improved by hardware modifications at architectural level. The method to realize this is to modify the behavior of the cache–which is a bridge between processor and main memory–and make it random. Mezzetti *et al.* [78] show that time-randomized caches bring several benefits to hard RT system: it reduces user's efforts for timing analysis and provides tight WCET.

The behavior of a cache is determined by two policies: replacement policy and placement policy, and they are made random respectively for pWCET analysis.

For cache replacement, every time a new memory request comes into the cache set from the main memory, one cache block in this set will be selected and evicted. The new address is put into the position of the evicted block. There are several replacement policies for conventional caches, such as First-In-First-Out (FIFO), Least Recently Used (LRU), Most Recently Used (MRU) [8], etc. To make the cache behavior random for a new memory address, one can adopt random replacement policy . When using a random replacement policy, every time the cache eviction happens, a cache block is selected randomly to be replaced by the new memory request. Quinones *et al.* [90] study a random replacement policy for standard and skew-associate caches and they compare simulation results with caches using the LRU replacement policy, because it performs best in terms of predictability [93]. The authors show that caches with random replacement policy reduce performance anomalies. For example, in one case study, the hit ratio of a cache using the LRU is from 0.41 to 0.93; while for a cache with random replacement policy, it varies from 0.64 to 0.94.

The cache placement policy has an impact on cache behavior as well. For cache placement, when choosing the cache set, a conventional cache uses several bits of the memory address. Schlansker *et al.* [96] propose a random placement policy and investigate its impact by matrix operations. This policy distributes cache entries more uniformly, and cache miss ratio is lower than that from conventional caches. Topham and Gonzalez [107] use a random placement policy and the results show that it can reduce cache conflicts and improves system

performance. However, the random placement policy in [96, 107] adopts a pseudo-random hash function that depends on memory addresses. Hence for a given memory layout, it always produces the same placement distribution. Kosmidis *et al.* [66] propose a cache with a random replacement policy and a parametric random placement policy, which requires little overhead in terms of complexity and energy consumption. The introduction of the parameter into the hash function ensures that the placement distribution is randomized for the same memory layout, so that it is feasible to apply probabilistic timing analysis.

In addition to aforementioned hardware techniques, software techniques (e.g. compiler and runtime techniques) can also be applied to make a system behavior random. Berger and Zorn [16] present DieHard, a runtime system, to allocate memory randomly. The *probabilistic memory safety* is achieved by using a large heap space. The DieHard manager deals with objects in the heap and reduces memory error probabilities. Kosmidis *et al.* [67] propose a software approach to randomize the behavior of conventional caches for use with probabilistic timing analysis. This work modifies code and data memory objects offline using compiler and linker. When the program starts or objects are allocated, the memory objects are placed in random locations by dynamic randomization code in the executable. Dynamic randomization challenges safety requirements (e.g. ISO26262) in the automotive domain. To solve this issue, Kosmidis *et al.* [64] present a static software randomization method. Several binary files are produced for the same program, in which memory objects are created with offsets to achieve random effects. Kosmidis *et al.* [65] develop a software tool to modify source code of the program, which realizes randomization without modifying existing toolchains.

In pWCET analysis, the result is expressed in terms of a density function or an exceedance function: it shows the probability of an application for given execution or the probability to exceed a given execution time. This can further be classified into three categories: Measurement-Based Probabilistic Timing Analysis (MBPTA), Static Probabilistic Timing Analysis (SPTA) and the combination of both methods.

In MBPTA, execution time measurements are collected and predictions are made using Extreme Value Theory (EVT). EVT [41, 13] is a statistical methodology that studies extremely rare events (i.e. events at the tails of the distribution) that may have severe consequences, when little experimental evidence is available. Usually two methods can applied to EVT: Block Maxima (BM) [37] and Peaks Over Threshold (POT) [17].

In [26, 44], Burns and Edgar demonstrate how to predict execution time with measurements by an EVT method. Raw data are fit to the Gumbel distribution [51] and results are represented as a density function with respect to execution time. Hansen *et al.* [53] explain why raw data fitting in [44] is incorrect. A BM method using EVT for WCET distribution esti-

mation is thus presented. Griffin *et al.* [48] investigate assumptions required by the Gumbel distribution, and study precision sacrificed due to this statistical method. Additional restrictions on the EVT method are proposed for safe applications. Lu *et al.* [75] propose a new way of sampling mechanism to estimate program execution time on a single processor and EVT is combined with this sampling technique. A more recent work using EVT is from [37]. A BM method is applied to fit the Gumbel distribution using a quantile plot, needing only a few hundred simulation runs. This significantly reduces the number of required measurements. Kosmidis *et al.* [63] study how processor architectures should be modified to meet MBPTA requirements. Wartel *et al.* [111] apply MBPTA to real avionics applications and results show tight pWCET estimates. Abella *et al.* [5] investigate when MBPTA fails due to pathological cases and propose *Heart of Gold* techniques to detect these cases. Lesage *et al.* [72] introduce a framework for MBPTA result evaluation. Synthetic tasks are used to provide realistic data and actual WCET can be computed using proposed framework.

Apart from statistical analysis of measurements, another way of PTA is Static PTA (SPTA). This requires detailed knowledge of software and hardware. A timing bound can be obtained by theoretical analysis: with the given assumptions, the SPTA method analyzes instruction or data caches and obtains a probabilistic distribution of the program's execution time.

Several works on SPTA have been proposed for caches with random replacement policy. Zhou [118] proposes a cache hit formula using reuse distance–the number of memory addresses accessed between two consecutive references to the same memory address–which simplifies computational complexity significantly. The probabilities for each cache access are made independent, and the final result is the convolution of all cache accesses. However, Cazorla *et al.* and Altmeyer *et al.* [27, 9] have found his methodology unsound. Quinones *et al.* and Kosmidis *et al.* [90, 66] give other formulea for evict-on-miss caches, and Cucu-Grosjean *et al.*, Cazorla *et al.* [37, 27] perform evict-on-access timing analysis using these formulea. However, Kosmidis *et al.* [66] may overestimate the cache hit ratio [40]. Thus, the result of probabilities for timing may be too optimistic and incorrect in this case.

Davis *et al.* [39] develop a formula using reuse distance only for evict-on-miss caches, and Altmeyer *et al.* [9] prove it to be optimal when only reuse distance is known. Multi-path programs are also analyzed by assuming that they are bounded. Besides, pre-emption impacts are taken into account for timing analysis. Altmeyer *et al.* [9] have proposed an exhaustive analysis approach for SPTA. To reduce the computational complexity, the exhaustive approach can be combined with simplified formulae. This approach is improved in [11], with an improved algorithm for SPTA. Griffin *et al.* [49] propose a methodology from the field of Lossy Compression and they use a fully-associative cache for timing analysis throughout

their work. By using *May* and *Must* Analysis, the result is more accurate with appropriate parameters. To demonstrate the impact of time-randomized caches, Reineke, Abella *et al.* and Altmeyer *et al.* [92, 4, 11] have done comparisons between caches using LRU and random replacement policy. Lesage *et al.* [71] develop an SPTA for multi-path programs. A worst-case execution path is obtained using a joint function by exploring cache states and path inclusions. Based on SPTA from [9], the pWCET can be calculated.

Hybrid analysis combines both MBPTA and SPTA for timing behavior prediction. So far, very little research has been done for hybrid timing analysis. Bernat *et al.* [18, 20] introduce a hybrid timing analysis for probabilistic hard RT systems: an RT program is analyzed and its structure is represented as a syntax tree. Instrumentation and trace are generated, so that distributions of all blocks can be produced locally using SPTA or MBPTA. A traversal of the syntax tree is used to calculate the WCET of the program. Due to dependencies between different blocks, copulas is proposed by [21] to obtain a lower bound.

In this paper, we propose an adaptive Markov chain based SPTA methodology for time-randomized caches. Rather than using reuse distance only, this method adopts more information and produces more accurate results. By limiting number of states used in the Markov chain model, the computational complexity has been restrained to make the calculation feasible.

7.5 System Model

In this section, we present a methodology to model the timing of a system with a probabilistic cache using state space model. The next state of the system with random replacement caches solely depends on current state, which satisfies the Markov property and can be represented as a Markov chain. A set-associative cache is used as an example, but note that a direct mapped cache can be seen as a special case of set-associative cache, in which the associativity equals 1; a fully-associative cache is another special case, in which the associativity equals the number of available cache blocks.

7.5.1 Cache Architecture

A set-associative cache is shown in Figure 7.1. This cache has several sets, and for each of those it provides a number of ways to store cache blocks. Each memory address used by the cache is divided in three parts: *tag* bits, *set* bits and *offset* bits. *offset* bits locate the data within each cache block, *set* bits are used to find which cache block should be selected for a given memory address. As multiple memory addresses can be stored in the same cache block,

Memory Address: | *tag* | *set* | *offset* |

	Way 1	Way 2	Way 3	Way 4
Set 0				
Set 1				
Set 2				
Set 3				

Figure 7.1 Set-associative cache representation

tag bits are stored within each cache block for comparison to identify the correct memory address it refers to. There exist several cache policies that describe how addresses are placed and replaced in the cache. In this paper, we consider a cache with modulo placement policy and evict-on-miss random replacement policy.

For the modulo placement policy, the *set* bits are used to select the cache set in which the data will be stored using the modulo operation. With an evict-on-miss random replacement policy, every time a cache miss happens, a way is selected randomly, and the cache block data are replaced by the new memory content.

To calculate the timing distribution of a probabilistic system, we first obtain the timing distribution of each cache set and the timing associated with the whole cache can be obtained by performing a convolution across all sets. Since modulo placement policy is adopted, memory addresses in different cache sets are stored separately and do not affect each other, i.e. the memory address in one cache set does not change the hit or miss probabilities in another cache set. Hence they are independent of each other statistically. As a result, the final timing distribution can be obtained using convolutions.

7.5.2 State Space Exploration

Let us assume there are distinct memory addresses $M = \{a, b, c, \cdots\}$ that are allocated to one cache set. The state space $\mathbb{S}$ can be constructed in a way such that the combinations of the distinct memory addresses are elements of the state space. The element $s_i \in \mathbb{S}$ is the state of the system, and it represents a unique cache configuration, i.e. the memory address layout of the system. The state space $\mathbb{S}$ is formally defined as:

$$\forall A \subseteq M, A \in \mathbb{S}$$

Let N_w be the number of ways (i.e. associativity) of the cache set. Then we have $|s_i| \leq$

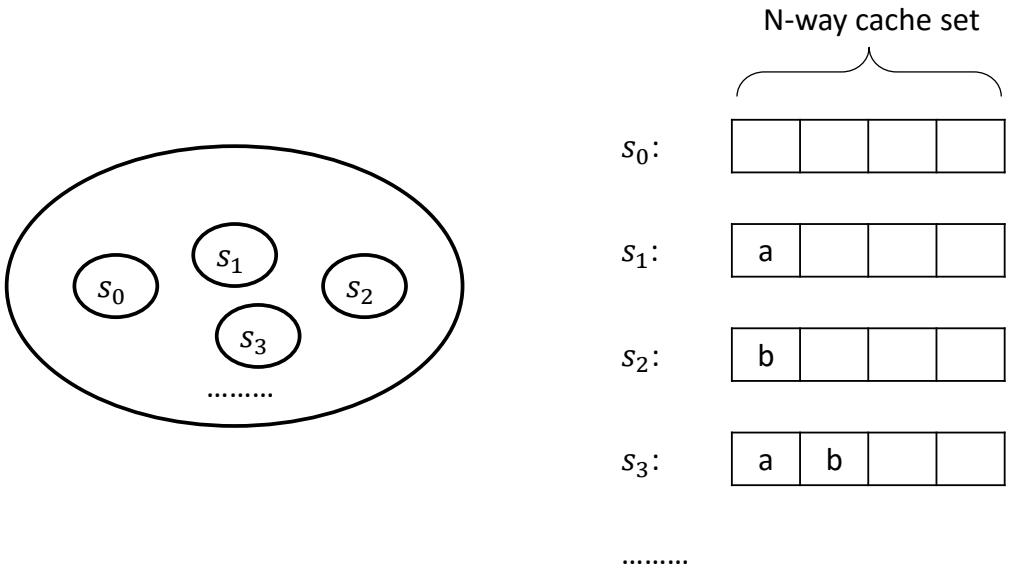


Figure 7.2 State space exploration

$min(N_w, |M|)$, i.e. the number of distinct memory addresses in a state is less than or equal to the minimal value between cache associativity and the number of distinct memory addresses for this set.

Figure 7.2 is used as an illustration of state space construction, in which we define states $\mathbb{S} = \{s_0, s_1, s_2, s_3, \cdots\}$ that correspond to the following configurations:
$s_0 = \emptyset$: empty cache
$s_1 = \{a\}$: address a in cache
$s_2 = \{b\}$: address b in cache
$s_3 = \{a, b\}$: address a, b in cache
$\vdots$
A cache memory is generally organized in blocks of more than one byte. In the following analysis the term address refers to the block identifier, i.e. the tag address.

Given this state representation, one set of a time-randomized cache can be modeled with the following Markov chain:

$$S_n = S_{n-1} \cdot P_{n-1} \tag{7.1}$$

where S represents state occurrence probability vector for the cache and P represents the transition matrix which determines how the current state probability S is transformed into a new state S. S_n and P_n represent the state probability vector and transition matrix at step n. We assume that initially there are no memory addresses in the cache, i.e. the system starts executing with an empty cache. Let us suppose that there are N states, i.e. $|\mathbb{S}| = N$,

then the state probability vector is

$$S = [Pr(s_0), Pr(s_1), \cdots, Pr(s_{N-1})] \tag{7.2}$$

where $Pr(s_i)$ is the probability of state s_i.

The number of states N in S_n is

$$N = \sum_{k=0}^{l} \begin{pmatrix} |M| \\ k \end{pmatrix} \tag{7.3}$$

with $|M|$ the number of addresses associated with the set, and $l = min(N_w, |M|)$ i.e. the minimal value between cache associativity and $|M|$.

From Equation (7.3), we can see that the number of states N is a function of cache associativity N_w and the number of addresses $|M|$ for one set. For a specific cache, as the number of memory addresses $|M|$ increases, the computational complexity increases polynomially with a large exponent $|M|$, which becomes eventually intractable for computation. However, the cache organization in blocks helps reducing the total number of memory addresses. Besides, as cache size increases, the probability for all code and data to reside in one cache set decreases, which effectively lowers the value of variable $|M|$.

7.5.3 Transition Matrix Calculation

Every time a new memory address is accessed, the cache state may change, and the transition matrix for next step varies accordingly. Therefore, we need a non-homogeneous Markov chain model, i.e. a Markov chain model whose transition matrix varies over time. The way to compute the transition matrix at each step is demonstrated in this section.

The transition matrix is represented as

$$P = \begin{pmatrix} p_{0\to0}, & p_{0\to1}, & \cdots, & p_{0\to N-1} \\ p_{1\to0}, & p_{1\to1}, & \cdots, & p_{1\to N-1} \\ \vdots & \vdots & \ddots & \vdots \\ p_{N-1\to0}, & p_{N-1\to1}, & \cdots, & p_{N-1\to N-1} \end{pmatrix} \tag{7.4}$$

where $p_{i\to j}$ is the probability for the system to go from state s_i to state s_j. In our model, the probability $p_{i\to j}$ varies constantly depending on current system state and the transition matrix thus needs to be computed at each time step.

Algorithm 2 shows how to compute the transition matrix. It takes two inputs (state s_i and

ALGORITHM 2: Transition matrix calculation

Data: State s_i, memory address a

Result: Transition matrix element $p_{i \to j}$

```
1  N_w ← cache associativity;
2  if a ∈ s_i then
3  |  p_{i→i} ←1; //cache hit
4  end
5  if a ∉ s_i then
6  |  //cache miss
7  |  for b ∈ s_i and s_j=s_i \ {b} ∪ {a} do
8  |  |  p_{i→j} ←1/N_w; //one address is replaced
9  |  end
10 |  if |s_i| < N_w then
11 |  |  for s_j=s_i ∪ {a} do
12 |  |  |  p_{i→j} ←(N_w − |s_i|)/N_w; //one address is added
13 |  |  end
14 |  end
15 end
```

the incoming memory address a) and produces one output (transition matrix element $p_{i \to j}$). The algorithm checks state s_i and generates the transition matrix elements accordingly as follows:

Line 2: If the requested memory address is in the state ($a \in s_i$), there is a cache hit. In this case, the cache will not change its state and it thus has a probability of 1, i.e. $p_{i \to i} = 1$.

Line 5: If the requested memory address is not in the state ($a \notin s_i$), there is a cache miss and the transition matrix is computed. This is the most complex case: the new memory address may replace an existing cache block, or it may be put into a new cache block and probabilities have to be computed accordingly. In our target cache, the probability of replacing an existing cache block is $1/N_w$ (see Line 8), where N_w is the cache associativity. This is because we consider an evict-on-miss time-randomized cache, and a cache block is randomly selected for replacement with probability $1/N_w$. The probability for a memory address to be placed in an empty cache block is $(N_w - |s_i|)/N_w$ (see Line 12), where $|s_i|$ is the number of blocks in use for the current state s_i. This is due to the fact that if the new memory address does not cause a replacement, it can only be put into an empty cache block. The number of empty cache blocks is $N_w - |s_i|$, and they are chosen from N_w ways. Therefore the probability is $(N_w - |s_i|)/N_w$. Example 7.5.1 is given for illustration.

Example 7.5.1 *Suppose we have a cache set with associativity of 4. The first and second*

cache blocks have been used, and the third and fourth cache blocks are empty. When a new memory address is cached, the probability of going into the first cache block is $1/N_w$*, i.e. 1/4. Similarly, for the second cache block, the probability is 1/4. They are computed separately, because they have different addresses, which represent different states* s_i*. The probability of loading a block into an empty cache blocks is* $(N_w - |s_i|)/N_w$*, i.e. 1/2. Since the third and fourth cache blocks are both empty, loading into one or the other has the same effect, and therefore it represents a single state considering the probability of both.*

Using Algorithm 2, Equation (7.1) can be used to describe state transitions of the system, provided the initial distribution S_0 is known. However, the cumulative timing information, which shows timing with respect to probability, is still unknown. To solve this issue, Section 7.6 introduces the vector that stores timing information for each state.

7.6 Timing Analysis

To describe the timing behavior of a system, we employ a vector containing timing information. This vector–together with the state space model described in Section 7.5–can describe the system timing behavior, where the state space model specifies occurrence probabilities of all states, and the timing vector specifies how the execution time is distributed for each state.

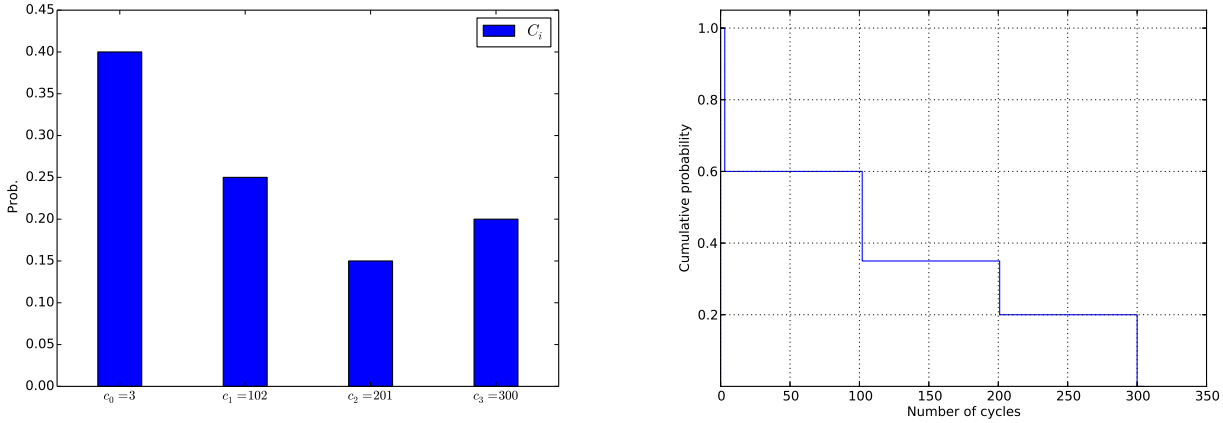
7.6.1 Timing Representation

A vector C_i can be used to denote the timing distribution in terms of number of cycles for a state s_i, and a vector CP_i can represent the probability of occurrence for C_i. Note that the number of cycles is different from the time step used in the Markov chain. At each time step, one memory address is accessed and different number of cycles may be applied to the timing analysis according to the system status (e.g. 1 cycle for a cache hit and 100 cycles for a cache miss). Then we have

$$C_i = [c_0^i, c_1^i, \cdots]$$

$$CP_i = [Pr(c_0^i), Pr(c_1^i), \cdots]$$

where c_j^i represents the program duration in cycles in ascending order for state s_i and $Pr(c_j^i)$ the occurrence probability for c_j^i. As an example, Figure 7.3(a) shows the timing distribution in C_i and CP_i for state s_i. One can see that the probabilities for a duration of 3, 102, 201 and 300 are 0.40, 0.25, 0.15 and 0.20 respectively.



(a) Timing distribution

(b) Cumulative timing

Figure 7.3 Timing analysis example

7.6.2 SPTA Convolution

For set-associative caches, we use convolutions to get the timing of different cache sets. In probability theory and statistics, if two random variables are independent, then the sum of them follows a distribution which is the convolution of both distributions. By using modulo placement policy, we make sure that memory addresses in one cache set do not affect hit or miss probabilities of memory addresses in another cache set. Therefore independence is guaranteed. We show the convolution in detail by using Execution Time Profile (ETP).

The Execution Time Profile (ETP) is used to represent timing information and its associated probability. The ETP for state s_i is defined as

$$ETP_i = \{C_i, CP_i\} = \{[c_0^i, c_1^i, \cdots], [Pr(c_0^i), Pr(c_1^i), \cdots]\}$$

where C_i is the timing distribution vector, and CP_i is its corresponding occurrence probability vector.

Suppose there are two ETPs: ETP_i and ETP_j, and their convolutions is ETP_k. Then the convolution is performed as follows (the symbol $*$ is used as the convolution operator)

$$\begin{aligned} ETP_k &= ETP_i * ETP_j = \{C_i, CP_i\} * \{C_j, CP_j\} \\ &= \{[c_0^i, c_1^i, \cdots], [Pr(c_0^i), Pr(c_1^i), \cdots]\} * \{[c_0^j, c_1^j, \cdots], [Pr(c_0^j), Pr(c_1^j), \cdots]\} \\ &= \{C_k, CP_k\} = \{[c_0^k, c_1^k, \cdots], [Pr(c_0^k), Pr(c_1^k), \cdots]\} \end{aligned}$$

where

$$c_l^k = c_m^i + c_n^j \tag{7.5}$$

$$Pr(c_l^k) = \sum_{c_l^k = c_m^i + c_n^j} Pr(c_m^i) Pr(c_n^j) \tag{7.6}$$

The convolution of two ETPs is demonstrated in Example 7.6.1. Two ETPs–ETP_1 and ETP_2–are provided. ETP_1 has three possible timing values: $[1, 2, 3]$ and their probabilities are $[0.1, 0.3, 0.6]$; the timing distribution and corresponding probabilities for ETP_2 are $[1, 3]$ and $[0.2, 0.8]$ respectively. From Equation (7.5), we can see the element in the new timing distribution is the sum of timing distributions of elements in ETP_1 and ETP_2, i.e. $[2, 3, 4, 5, 6]$. The new probability vector is calculated using Equation (7.6). Its element is the sum of product of two elements in probability vectors of ETP_1 and ETP_2, provided the sum of corresponding timing distributions are the same. For example, for the cycle 4, its corresponding probability is 0.2, which consists two parts: the first part is the sum of cycle 1 in ETP_1 and cycle 3 in ETP_2. The second part is the sum of cycle 3 in ETP_1 and cycle 1 in ETP_2. So the corresponding probability is $0.1 \times 0.8 + 0.6 \times 0.2 = 0.2$.

Example 7.6.1

$$ETP_1 = \{[1, 2, 3], [0.1, 0.3, 0.6]\}, ETP_2 = \{[1, 3], [0.2, 0.8]\}$$

$$\begin{aligned} ETP_1 * ETP_2 &= \{[1, 2, 3], [0.1, 0.3, 0.6]\} * \{[1, 3], [0.2, 0.8]\} \\ &= \{[2, 3, 4, 5, 6], [0.02, 0.06, 0.2, 0.24, 0.48]\} \end{aligned}$$

7.6.3 Cumulative Timing

Having the timing information for each cache set, we can compute a cumulative timing plot. This gives us an exceedance function, showing the probability of exceeding a certain program duration in cycles. The timing distribution is the same as from the timing vector, i.e. C_i. The cumulative probability is denoted as CPC_i, and its jth element is calculated as follows

$$CPC_i[j] = \sum_{CP_i[k] > CP_i[j]} CP_i[k] = \sum_{Pr(c_k^i) > Pr(c_k^j)} Pr(c_k^i) \tag{7.7}$$

where n is the number of elements in CP_i.

As an example, the cumulative probabilities to exceed a program duration of 3, 102, 201 and 300 are 0.6, 0.35, 0.2 and 0 respectively. The result is plotted in Figure 7.3(b).

7.6.4 Timing Integration

From previous sections, it can be seen that timing vectors can be used to express the timing behavior of a system. In this section, we discuss how to integrate timing vectors into our state space model based on a non-homogeneous Markov chain model.

Since we use a Markov chain model for our system, at every step the system can be described by the states in the state space. To calculate timing information, timing vectors are integrated into the Markov chain model. Each state s_i is assigned a vector C_i to keep its timing. The timing vector may expand as time goes on, since more duration may appear as the system state evolves. In addition, a corresponding vector CP_i–that represents the occurrence probabilities of each possible timing–is generated at the same time. The algorithm to calculate timing is illustrated in Algorithm 3.

ALGORITHM 3: Calculate timing distribution

Data: Transition matrix P, timing distribution C, corresponding prob. CP
Result: New timing distribution $C2$, prob. $CP2$

```
1  N_h ← cycle number for cache hit;
2  N_m ← cycle number for cache miss;
3  C ← ∅;
4  CP2 ← ∅;
5  for c_k^i ∈ C_i and c_k^j ∈ C_j do
6      if p_{j→i} ≠ 0 then
7          if i=j then
8              //cache hit
9              c_k^i = c_k^i + N_h;
10         else
11             //cache miss
12             c_k^i = c_k^j + N_m;
13             Pr(c_k^i) = Pr(c_k^j) · p_{j→i};
14         end
15     end
16 end
17 C2 = C2 + C;
18 CP2 = CP2 + CP;
19 //merge timings with the same cycle number
20 Merge C and CP;
```

Algorithm 3 takes three inputs (transition matrix P, timing distribution vector C and its corresponding probability vector CP) and produces two outputs (new timing distribution

vector $C2$ and its corresponding probability vector $CP2$). We can see that timing information is associated with the Markov chain model transition matrix P. All values of elements in P are examined: if not 0, this means that the system will change its state, and the timing vectors will be used to compute the timing associated with the transition. There are 2 cases for timing calculation, as seen below

Line 7: If the element of transition matrix P to be examined is on the diagonal, then it is $p_{j\to i}$, where $j = i$, i.e. the state does not change and the access is a cache hit. In this case, timing vectors will be expanded: the duration for a cache hit (e.g. 1 cycle) is added to all elements to timing vector C_j which keeps the timing of state s_j and the result is integrated into timing vector C_i for s_i. The corresponding probability vector CP_j is directly integrated for CP_i, since a cache hit forces the transition probability $p_{i\to i} = 1$.

Line 10: If the transition matrix element is not on the diagonal, then it is $p_{j\to i}$, where $j \neq i$. It means the state has changed from state s_j to s_i and therefore it is a cache miss. In this case, the timing vectors will be expanded the same way as in the previous case: the duration for a cache miss (e.g. 100 cycles) is added to C_j and integrated into C_i. However, its corresponding probability CP_j is multiplied by the transition probability $p_{j\to i}$. The probability result considering each transition is integrated into CP_i.

With the Algorithm 3, new timing vectors are generated based on old ones. However, in the new timing vectors there may be duplicate duration. Such mutually exclusive cases should be merged: timings with the same duration are merged by adding their probabilities. For example, there is a pair of timing and probability vectors

$$C_i = [100, 100, 201], CP_i = [0.2, 0.7, 0.1]$$

can be merged as

$$C_i = [100, 201], CP_i = [0.9, 0.1].$$

7.6.5 Analysis Framework

By computing timing vectors together with the state space model, we can obtain the required timing information. The framework of our computation is displayed in Figure 7.4.

In this framework, a transition matrix P_{n-1} is computed at every step. With the transition matrix P_{n-1}, the new state S_n is obtained using a Markov chain matrix model, which will

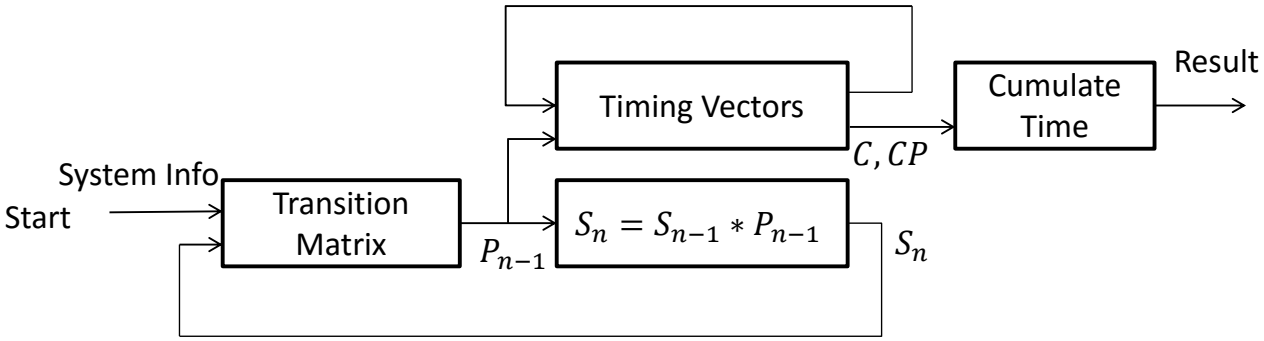


Figure 7.4 Markov chain framework

be used for the transition matrix calculation for the next step. Meanwhile, timing vectors C_i, CP_i are generated using the transition matrix. These are fed back to the next step. Finally, timing vectors are accumulated to form the timing exceedance function.

7.7 Adaptive Method

In previous sections, we have demonstrated how to use a Markov chain based model to do timing analysis. The result of this method is accurate, but since the number of states increases polynomially with a large exponent, this method is intractable. As a result, we introduce an adaptive method to limit the number of states and to produce a result with reasonable accuracy.

7.7.1 State Modification

There are different ways to select some memory addresses for Markov chain states. Here we propose an adaptive method and it replaces states in the markov chain continuously.

We have shown that state s_i is used to represent a unique memory layout of the system. Suppose there are $|M|$ memory addresses, to reduce computational complexity, we would like to use only n ($n < |M|$) memory addresses to represent states. This is realized using state modification in two steps:

- State construction: for the first n addresses, we construct the state space using the Markov chain method as $\{s_0, s_1, ...\}$. This way, the number of states does not increase polynomially any more- it is limited to the value given by Equation (7.3). When another new memory address comes, we modify the memory addresses in state s_i in next step, instead of increasing the number of states.

- State modification: when memory address a is accessed, we first check if it is already in state s_i. If $a \in s_i$, it means this memory address is already in our state space. In this case, we do not need to modify the state. If $a \notin s_i$, it means this is a new address, and we would like to modify our state such that this address a is included in the state space. Meanwhile, one memory address in the state space will be discarded to have the same number of address. To modify the state, we first find the memory address b that is not used in future, or whose next access takes the most time steps in all memory addresses consisting the state space when it is used. This method tries to keep all memory addresses that will be used shortly and discard those memory addresses that will be used after a long time. The state s_i containing b is then removed. The new memory address a is applied to the Markov chain model and new state s_j is constructed, i.e. $s_j = s_i \setminus \{b\} \cup \{a\}$. This way, the number of states remains the same, but the states represent different memory addresses: those states s_i containing address a have been replaced by states s_j containing address b.

We can see that we select a fixed number of addresses, which results in a fixed number of states. Besides, by looking into the future memory requests, we can find which addresses will be used shortly. Hence we are able to discard those addresses that will be used at a later time, while pessimistically merging their timing information into the existing states as described in Section 7.7.2.

7.7.2 Timing Analysis

When states are changed, we need to take timing analysis into account as well, because each state is assigned different timing distributions. To obtain the safety bound of pWCET, we use a conservative method. We need to deal with the following variables for timing analysis: state occurrence vector S, timing distribution vector C and its occurrence probability vector CP.

Suppose s_i is the state containing the memory address m_d to be discarded, and s_k is the state containing all other memory addresses in s_i except the address b, i.e. $s_k = s_i \setminus \{m_d\}$. In previous section, we see that when a new address is accessed, we may remove the state s_i. Therefore the state vector which represents its occurrence probability must be modified accordingly. In the new state vector S, we use this formula to modify it:

$$Pr(s_k) = Pr(s_i) + Pr(s_k) \tag{7.8}$$

Let m_i be the incoming address. For the new state $s_j = s_i \setminus \{m_d\} \cup \{m_i\}$, we have $Pr(s_j) = 0$.

Apart from the state vector modification, we need to modify the timing distribution vector C and its occurrence probability vector CP. This is performed in a similar way to the modification of the state occurrence vector S. Suppose C_i and CP_i are vectors for s_i, and C_k and CP_k are vectors for s_k. Then the modification is performed using formulae

$$C_k = C_k + C_i, CP_k = CP_i + CP_k \tag{7.9}$$

The elements with the same number of cycles are then merged, as shown in Line 20 of Algorithm 3.

After the modifications of the state occurrence vector S, timing distribution vector C and its occurrence probability vector CP, we have transformed our Markov chain model into a new one. The next address to be accessed is applied to this new Markov chain model by using Algorithm 2 and Algorithm 3 and timing analysis can thus be performed.

7.7.3 Safety of the Adaptive Method

We adaptively modify states using Equation (7.8) and (7.9), which can provide pessimistic and safe results. Hereby we explain why results are safe. Let s_n be a state without state modifications from the adaptive method, i.e. a state that contains all memory addresses. Let s_a be a state with adaptive method. Note that some memory addresses may be discarded by state modifications. Thus we have $s_a \subseteq s_n$. To study if the adaptive method using s_a produces safe results compared to the method using s_n, we need to compare C_a and CP_a with C_n and CP_n. Let m_i be the incoming address and we know that $C_a = C_n$ and $CP_a = CP_n$ before accessing m_i. We need to consider following cases:

- $m_i \in s_a$: it implies that $m_i \in s_n$. Consequently there is a cache hit for both s_a and s_n. From Algorithm 3, we can compute that $\forall p, C_a[p] = C_a[p] + N_h$, where $C_a[p]$ in the right-hand side is the element in C_a before accessing m_i, $C_a[p]$ in the left-hand side is the element in C_a after accessing m_i and N_h is the number of cycles for a cache hit. This is the same for C_n. Therefore after accessing m_i, we still have $C_a = C_n$ and $CP_a = CP_n$.

- $m_i \notin s_a$ and $m_i \in s_n$: for s_a, it is a cache miss; for s_n, it is a cache hit. In the case of a cache miss, a state s_a may become a state s_i, with associated C_i and CP_i. Using Algorithm 3, we have $\forall p, C_i[p] = C_a[p] + N_m$, where N_m is the number of cycles for a cache miss.

  When a cache miss happens, we need to take into account of all new states and all

states use the same C_i. We sum up the probability vector elements of all states. $CP_a[p] = \sum_i CP_a[p] \cdot p_{a \to i} = CP_a[p] \sum_i p_{a \to i} = CP_a[p]$. We can see that for a cache miss, the sum of probability vectors of new states is $CP_a = CP_n$. However, the element in C_a is larger than the element in C_n, because $N_m > N_h$. Therefore the result using s_a is safe and more pessimistic compared to the result using s_n.

- $m_i \notin s_a$ and $m_i \notin s_n$: we still have $C_a = C_n$ and $CP_a = CP_n$, but the element in C_a is added by N_m instead of N_h.

From previous discussion, we conclude that our method can provide a safe and pessimistic result. In our method, we have selected a memory address and have replaced it with incoming memory address.This address is selected to keep as much information as possible for the system: for future memory accesses, only the cache hits which are related to discarded memory addresses are ignored. Using this method, we take account of temporal characteristics of applications and make our model adaptive better to their dynamic changes, such as the case that cache locality changes during execution in terms of both cache contents and the active line number. In Section 7.9, we can see that this method increases result accuracy and reduces computational cost. Example 7.7.1 presents how to adaptively modify state space $\mathbb{S}$, state occurrence vector S, timing distribution vector C and the corresponding probability vector CP.

Example 7.7.1 *Suppose that the memory accesses are a, b, c, a, c, and they are accessed from step 1 to step 5. We limit the distinct memory address number $n = 2$. Besides, we assume that the cache associativity $N_w = 4$, the cache hit cycle $N_h = 1$, and the cache miss cycle $N_m = 100$.*

At step 1, before we access a, we construct the state space as $\mathbb{S} = \{s_0 = \emptyset, s_1 = \{a\}, s_2 = \{b\}, s_3 = \{a, b\}\}$. The state occurrence vector $S = [1, 0, 0, 0]$, timing distribution vector $C = [\emptyset, \emptyset, \emptyset, \emptyset]$, and the corresponding probability vector $CP = [\emptyset, \emptyset, \emptyset, \emptyset]$.

At step 3, we need to modify $\mathbb{S}$, S, C and CP. Before the modification, we have $\mathbb{S} = \{s_0 = \emptyset, s_1 = \{a\}, s_2 = \{b\}, s_3 = \{a, b\}\}$, with $S = [0, 0, 0.25, 0.75]$, $C = [\emptyset, \emptyset, [200], [200]]$ and $CP = [\emptyset, \emptyset, [0.25], [0.75]]$.

Then we change the state space to $\mathbb{S} = \{s_0 = \emptyset, s_1 = \{a\}, s_2 = \{c\}, s_3 = \{a, c\}\}$. b is replaced by c, because it is not used in following accesses. The associated vectors are changed accordingly: $S = [0.25, 0.75, 0, 0]$, $C = [[200], [200], \emptyset, \emptyset]$ and $CP = [0.25, 0.75, \emptyset, \emptyset]$. We can see that after state modification, we have $Pr(s_1 = \{a\}) = 0.75$, which is a pessimistic case of $Pr(s_1 = \{a, b\}) = 0.75$. In the same way, we change the state from $Pr\{b\} = 0.25$ to $Pr\{\emptyset\} = 0.25$. The corresponding C and CP are modified accordingly.

7.8 Extension to Data Caches

We have demonstrated our Markov chain approach that can be applied to instruction caches directly. However, writing policies make data caches behave differently from instruction caches. In this section, we explain how to extend our approach to data caches.

7.8.1 Data Cache Writing Policies

There exist two writing policies for data caches: write-through and write-back policies. A write-through policy writes data to both the cache and the main memory at the same time. This can be modeled very easily and our Markov chain method can be applied in a straightforward fashion.

However, a write-back data cache behaves differently and it is often combined with a write-allocate policy, as illustrated in Figure 7.5. When reading or writing data from a write-back cache, we need to check if the selected cache block is set as 'dirty'. If so, the data in the cache block must be sent to the main memory first, because it has been modified, which results in an additional latency.

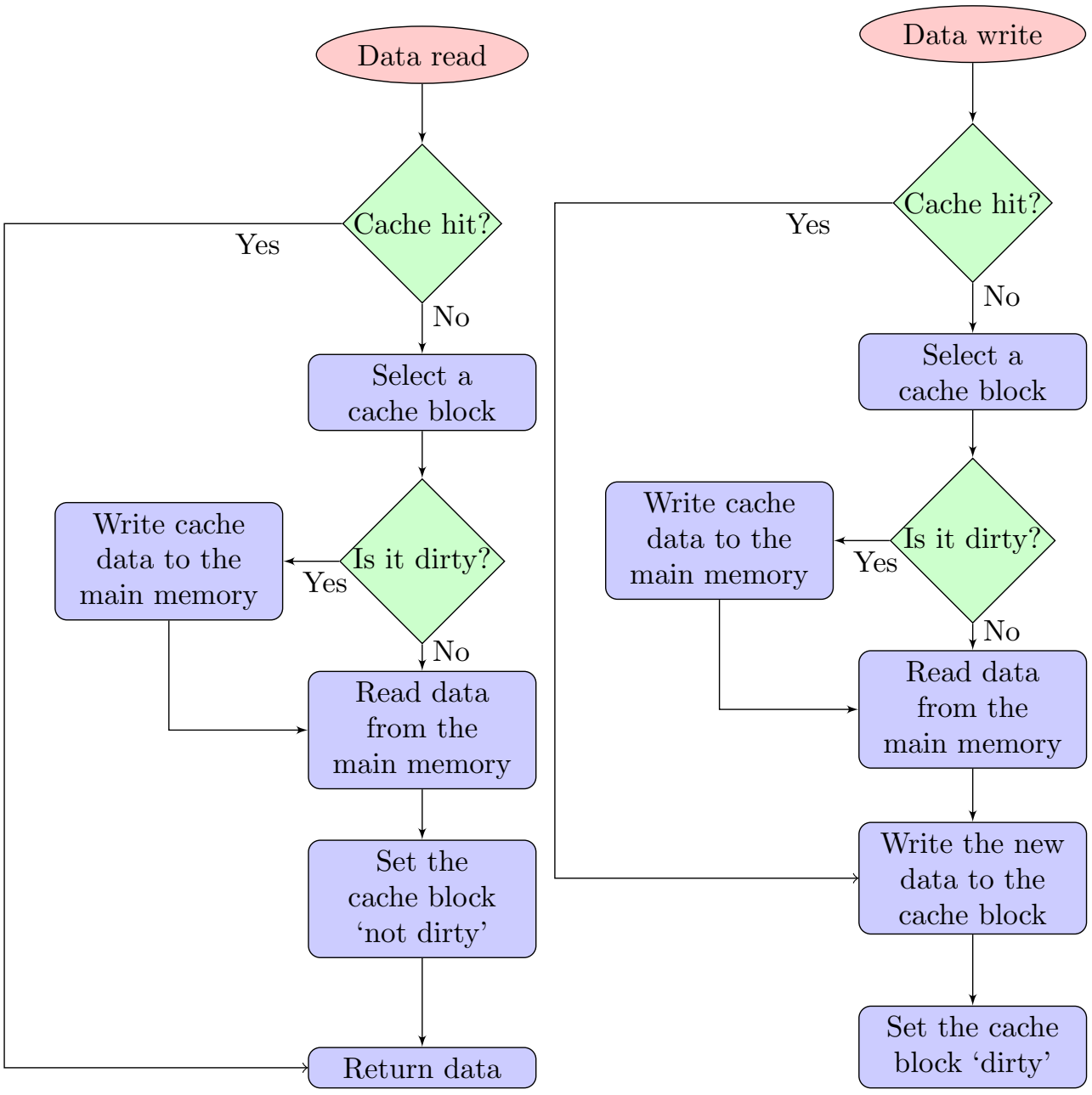
To extend our approach to data caches, we assume that when reading or writing, a cache access latency is N_h, and a main memory access latency is N_m. This is the same as what we used for instruction caches. From Figure 7.5, we can calculate latency L for the following scenarios:

- Cache hit: $L = N_h$.
- Cache miss and 'not dirty': $L = N_m$.
- Cache miss and 'dirty': $L = 2N_m$.

7.8.2 Method Modification

Compared to instruction caches, there is an additional latency N_m for cache misses in the presence of 'dirty' cache blocks. Thus we need to modify Algorithm 2 and 3. We introduce binary variables $B_d^a, B_t^a \in \{true, false\}$ to represent if the cache block with address a is dirty and the type of the data access to a, respectively. When the block with address a is dirty, $B_d^a = true$; otherwise $B_d^a = false$. If it is a data write, $B_t^a = true$; otherwise $B_t^a = false$.

First, we add additional operations to Algorithm 2 as follows:



(a) Read data from a data cache.

(b) Write data to a data cache.

Figure 7.5 Read and write for a write-back data cache with a write-allocate.

Line 5: In the case of a cache miss, we set the block as 'not dirty' if it is a data read, i.e. if $B_t^a = false, B_d^a = false$.

Line 15: We set the block as 'dirty' if it is a data write, i.e. if $B_t^a = true, B_d^a = true$.

Next we modify Algorithm 3 to account for additional latencies due to dirty blocks as follows:

Line 7: We modify the case of a cache hit, such that if a block is not dirty, the latency is N_h; otherwise it is N_m. Then we remove Line 9 and add the following: if $B_d^a = false, c_k^i = c_k + N_h$; otherwise $c_k^i = c_k^i + N_m$.

Line 10: We modify the case of a cache miss in a similar way. If a block is not dirty, the latency is N_m; otherwise it is $2N_m$. Note that when using the adaptive method, some addresses may be discarded, which results in empty cache blocks. We use $B_{discard}^a = true$ to indicate the address a has been discarded before, and $B_{discard}^a = false$ otherwise. When we access discarded addresses in future, we assume pessimistically that the blocks with such addresses are dirty. Consequently, we remove Line 12 and add the following: if $B_d^a = false \wedge B_{discard}^a = false, c_k^i = c_k^j + N_m$; otherwise $c_k^i = c_k^j + 2N_m$.

By adding operations related to dirty blocks to Algorithm 2, we are able to tell if each cache block is dirty or not. We modify Algorithm 3 to account for additional latencies caused by dirty blocks, and use pessimistic assumptions while applying the adaptive method.

7.9 Benchmarks Evaluation

In this section, we evaluate our methodology using real-world benchmark applications. We chose the Mälardalen benchmarks [52], a popular benchmark suite used for WCET evaluation and analysis. We perform SPTA using our adaptive Markov chain model, and compare its results with results from another state-of-the-art SPTA methodology. All benchmarks are performed with a dual-core Intel Duo CPU running at 3.0 GHz with 4GB memory.

Our experiments use the SoCLib open platform[2] to simulate our design under test. SoCLib supports several processor architectures: we adopt the MIPS 32-bit processor architecture. The Mälardalen benchmark suite was compiled into MIPS ISA with the Sourcery CodeBench tool from Mentor Graphics[3]. The platform is equipped with a single MIPS processor with an L1 instruction cache, which has been modified to use evict-on-miss random replacement policy. Our experiments are performed for an instruction cache and a write-back, write-allocate cache.

For industrial and avionic embedded systems, cache associativity is usually fairly small. For example, the LEON3[4] processor has a configurable cache between 1 and 4 ways. Thus we set the cache size as 512 bytes, with 4-way associativity and 4-byte cache block. For each cache miss, we assume a duration of 100 cycles and thus a dirty data cache block causes another

[2] `http://www.soclib.fr/`

[3] `http://www.mentor.com/embedded-software/sourcery-tools/`

[4] `http://www.gaisler.com/index.php/products/processors/leon3`

100 cycles; for each cache hit, the delay is 1 cycle. Memory address traces are generated by the platform, which are used for SPTA and adaptive Markov chain model analysis. We considered modulo placement only.

Benchmarks[5] used for analysis are listed in Table 7.1. We select the benchmarks that do not require hard floating point unit that is absent in our SoCLib platform.

Benchmark	Description
expint	Series expansion for computing an exponential integral function
bs	Binary search
duff	Unstructured loop with known bound
statemate	Automatically generated code
fdct	Fast discrete cosine transform
jfdctint	Discrete cosine transformation
ndes	Bit manipulation, shifts, array and matrix calculations
compress	Data compression
edn	Vector multiplication and array handling
adpcm	Adaptive pulse code modulation
bsort100	Bubble sort
matmult	Matrix multiplication
fir	Finite impulse response filter

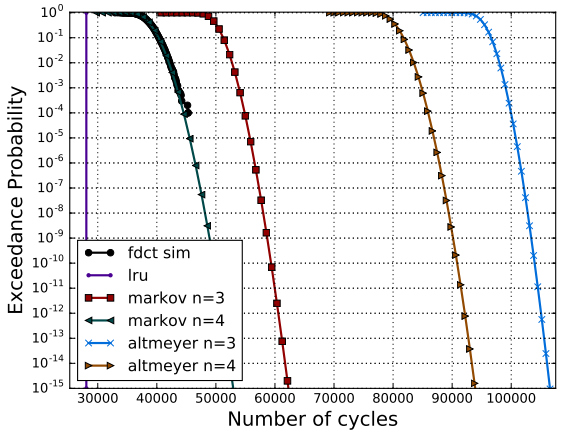
Table 7.1 Benchmarks

7.9.1 Model Accuracy

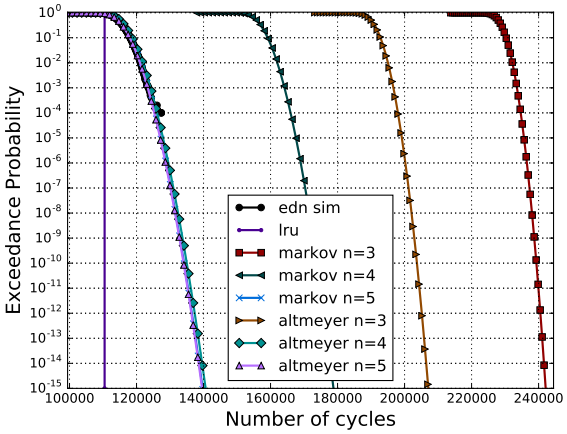
In this section, results from the adaptive Markov chain model are compared with simulations to verify its accuracy. For each cache set, different number of memory addresses are selected to see their impact on timing analysis. We select 10^{-15} as the exceedance probability of interest, since the maximum allowed failure rate is 10^{-9} per hour for commercial airborne systems, which is equivalent to an exceedance probability of around 10^{-13} [37]. Thus we estimate the time at 10^{-15} as a conservative result.

For the sake of space limitations, Figure 7.6 shows comparisons between simulations and a subset of the benchmarks using instruction caches and Figure 7.7 shows comparisons using data caches. Figure 7.6(a) and Figure 7.7(a) display the comparison between simulations and FDCT benchmark; Figure 7.6(b) and Figure 7.7(b) show the comparison for EDN benchmark. We ran 10,000 simulations to sample the timing behavior of the benchmark and performed cache analyses with the memory traces using our proposed approach. Simulated

[5] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

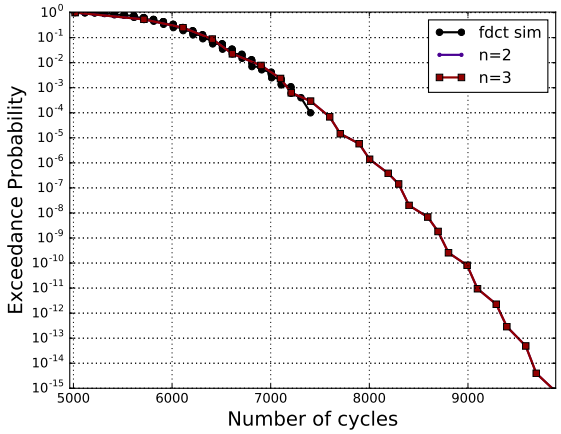


(a) Benchmark fdct. Total memory access: 1632. Distinct memory access: 267.

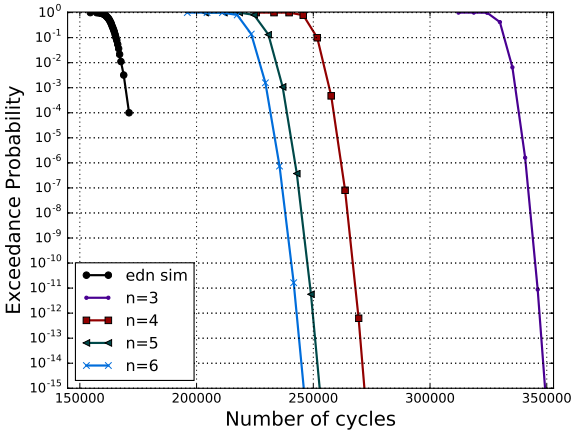


(b) Benchmark edn. Total memory access: 2398. Distinct memory access: 417.

Figure 7.6 Adaptive Markov chain model accuracy using instruction caches. A varying number of memory addresses are used in adaptive Markov chain model for comparison with simulations.



(a) Benchmark fdct. Total memory access: 364. Distinct memory access: 47.



(b) Benchmark edn. Total memory access: 11551. Distinct memory access: 524.

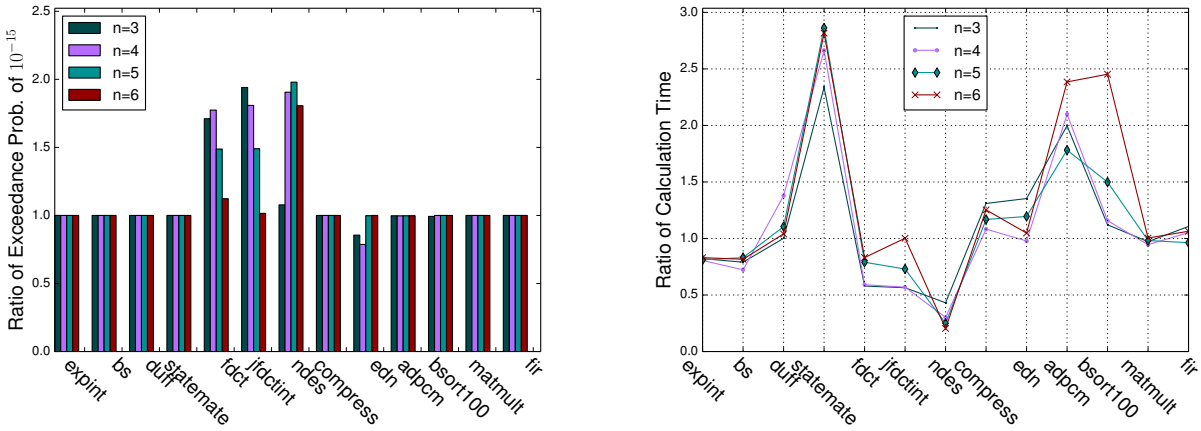
Figure 7.7 Adaptive Markov chain model accuracy using write-back data caches with write-allocate. Different number of memory addresses are used in adaptive Markov chain model for comparison with simulations.

time is obtained for each simulation. On each figure, the x-axis shows the number of cycles and y-axis represents the exceedance probability for corresponding cycles. This is called probabilistic WCET (pWCET), because for each WCET estimate, there is an associated exceedance probability. When we compare different results, we can see the WCET estimate

for a specific exceedance probability.

We can see as we increase the number of memory addresses n, the result from the adaptive Markov chain comes closer to that from simulations. Take Figure 7.6(a) for example, for $n = 3$ the number of cycles is 62,000 at the exceedance probability of 10^{-15}; for $n = 4$ it becomes 53,000, reducing the estimate pessimism by using more memory addresses. They will eventually produce the same result if all memory addresses are applied to the Markov chain method, and increasing the number n does not change the result anymore, as illustrated for $n = 4$ in Figure 7.6(a) and $n = 2, 3$ in Figure 7.7(a). In general, both the Markov chain methodology and simulations match well. Since simulations are performed randomly, there is a variance for each simulation, but the difference is not significant. However, as the probability goes down, fewer simulation samples are available, to the point where the results are not reliable: at the tail of the simulation plot, there is an obvious deviation between simulations and Markov chain methodology. This deviation is due to the lack of simulation samples, which were limited to constrain the simulation to feasible times. As the number of simulation samples increases, the simulation result converges to the result of our method. We thus conclude that our method can perform timing analysis accurately.

7.9.2 Comparison with Altmeyer SPTA



(a) Accuracy comparison.

(b) Time comparison.

Figure 7.8 Comparison with state-of-the-art SPTA. Accuracy and calculation time are compared respectively using different number of memory addresses.

In this section, we use instruction caches and compare the results from our methodology with that from the state-of-the-art SPTA proposed by [11], which is referred as "Altmeyer SPTA".

In "Altmeyer SPTA", memory addresses are divided into two independent parts: the state enumeration part and cache contention part. The addresses in the state enumeration part are used for a detailed analysis. A cache state is represented as a triple $CS = (E, P, D)$, where E contains memory addresses for the state, P is the probability of the state and D is the miss distribution. Every time an address is accessed, the update function is applied to update cache state. If the memory is not in the state enumeration part, update function evicts memory address in E; otherwise, the memory address is put into E. The probability P and miss distribution D are modified accordingly by the update function. By detailed analysis, a timing distribution can be generated.

To perform timing analysis for cache contention part, the notion of cache contention is introduced. All memory accesses are regarded as independent, and the lower bound probability of each memory access is calculated. Cache contention is used to see how memory addresses content for cache blocks. A simulation S is used to represent potentially conflicting addresses. If the accessed memory is not in S, the hit probability is 0. Otherwise, it is calculated using reuse distance, stack distance and cache associativity. This way, hit probabilities for all memory accesses are calculated and the timing distribution can be obtained by convolution. Since this part is independent of detailed analysis of state enumeration part, the convolution of timing distributions is the final distribution of the program.

The Altmeyer SPTA consists of two parts and the enumeration part uses the most-used memory addresses. This is different from our method, in which we use only one part, and we change the memory addresses in the state space adaptively.

In Figure 7.8, we compare the calculation accuracy and time of two different methods. The number of memory addresses for each cache set n ranges from $n = 3$ to $n = 6$. We start by using a small number of different memory addresses for each cache set $n = 3$. With such a number, some benchmarks show similar results to simulations, while others exhibit conservative timing predictions. As we increase memory addresses up to $n = 6$, most benchmarks have reached a point where further memory address increment improves the result accuracy slowly. The memory trace file sizes of the benchmarks are in ascending order from left to right in Figure 7.8.

Figure 7.8(a) shows the cycle number ratio between Altmeyer SPTA and the adaptive Markov chain based method. At the exceedance probability of 10^{-15}, we obtain estimated number of cycles using Altmeyer SPTA and the adaptive Markov chain model and they are represented as N_a and N_m, respectively. We calculate the cycle number ratio R_c as

$$R_c = \frac{N_a}{N_m}$$

We can see that when $R_c > 1$, the Altmeyer SPTA is more pessimistic; when $R_c < 1$, the adaptive Markov chain model is more pessimistic. Otherwise both methods produce the same result. On average, the geometric mean of Altmeyer SPTA estimates 11% more cycles than our adaptive Markov chain based method.

Figure 7.8(b) represents the time ratio between the Altmeyer SPTA and adaptive Markov chain based method. We use T_a to denote the calculation time for Altmeyer SPTA and T_m for the adaptive Markov chain model. The time ratio R_t is computed as

$$R_t = \frac{T_a}{T_m}$$

Figure 7.8(b) illustrates that the Altmeyer SPTA takes the approximately the same amount of time as our Markov model, with Altmeyer SPTA being 1% slower on average. The calculation time ratio varies within a limited range for all benchmarks (from 0.2 to 2.9), and overall the time difference between two methods is not statistically significant according to t-test at 95% confidence (p=0.22).

7.9.3 Comparison with LRU

In this section, we study impacts of cache on LRU replacement policy and random replacement policy. In Figure 7.9, we use *fdct* and apply different cache sizes and associativities. For a single-path program, the number of cycles is constant for caches with LRU policy. We use a simulation to obtain the number of cycles using LRU policy and plot it as a vertical line.

We can see that in Figure 7.9(a) and Figure 7.9(b), LRU performs worse than random replacement, i.e. the number of cycles from LRU policy is larger than that using random replacement policy. This is because for a smaller cache size, there are fewer cache sets. As a result, there are more memory accesses for each cache set. When the number of accesses becomes larger, the LRU performance becomes worse, since there are more opportunities to replace a cache block before its future use. Random replacement policy, however, is not affected so significantly. Each cache block is replaced randomly, which makes it possible to keep any cache block for future use. In the worst case, pathological case may occur for LRU caches, i.e. there are always cache misses for memory accesses in a cache set, since too many distinct memory addresses are used in such a pattern that they are evicted before their next accesses. A time-randomized cache can avoid such pathological cases since it evicts memory blocks randomly.

Figure 7.9(c) and Figure 7.9(d) illustrate that as the cache size increases, the number of cycles

decreases significantly, especially for LRU caches. On average, a larger cache size indicates fewer memory blocks for each cache set, which may avoid pathological cases effectively for LRU caches, which reduces number of cycles dramatically.

In addition, there are no associativity constraints on the use of our approach. In Figure 7.9(e) and Figure 7.9(f), we can see as cache associativity increases, our method can still be applied. The accuracy may be compromised, because more memory addresses are accessed when associativity increases. Some information is lost due to limited number of used blocks n, which compromises timing analysis result. The general rule is that n should be as large as possible, given the available computational resources.

Several previous studies have done comparisons between random and LRU replacement policies [100, 101, 90, 66]. Our experiments show that when the code size is larger than cache size, random policy helps reduce cache misses, which confirms the conclusion from previous studies that random policy can avoid pathological cases effectively. However, note that our results depend on the code layout of benchmarks and we analyze the cache impact using a particular code layout, i.e. the trace from the platform. This is only one of the entire code layout space. If the code layout changes, different results may be produced, because we have adopted set-associative caches with modulo placement policy in the experiments. The execution times for LRU and time-randomized caches may be different, i.e. they may be shorter or longer compared to the presented results, depending on changes of the code layout. In this section, we do not compare average performance of random and LRU policies, since we do not have memory traces of all code layouts.

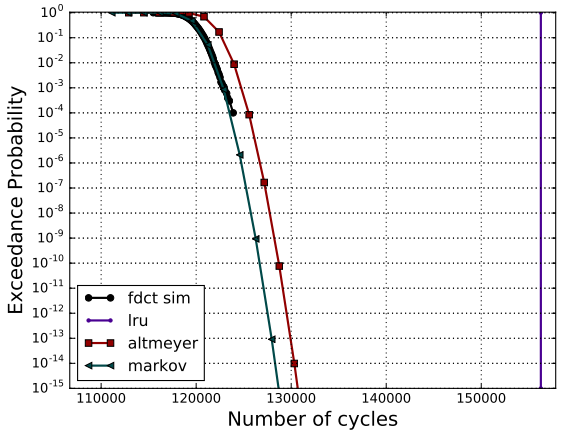
7.10 Conclusions

In this paper, we have demonstrated an adaptive Markov chain based Static Probabilistic Timing Analysis (SPTA) methodology. Our methodology is based on a non-homogeneous Markov chain model, which explores state space modeling for one cache set, and convolves different sets to generate final timing information. To reduce computational complexity, the state space can be limited to the specified level. The state space is modified adaptively, such that selected addresses can be replaced by new incoming addresses in the state space with good accuracy, while maintaining the same number of states. By reducing the number of addresses used for state modification, we can find a compromise between calculation accuracy and time.

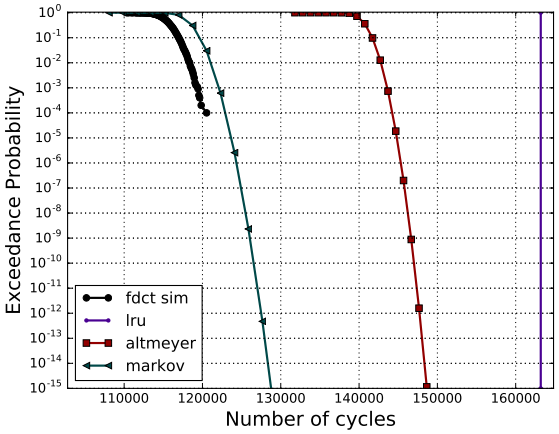
Benchmark applications are used to verify accuracy of this methodology by using simulations based on SoCLib platform with MIPS processor architecture. Its results are compared to

state-of-the-art SPTA methodology. It shows that with the adaptive state modification, our methodology has improved accuracy of results using less amount of calculation time. We also demonstrate how to evaluate cache impacts on system timing behaviors using the proposed method, which can help designers to select cache parameters of real-time embedded systems.

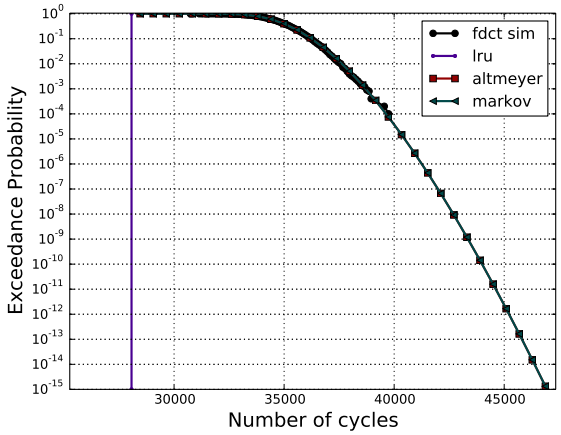
As future work, we can address several aspects: Simultaneous running of multiple programs and hybrid SPTA/MBPTA are two examples. In addition, we only explored single-path programs in this paper. However, it can be extended to multi-path programs by identifying the worst-case path. Fully extending the approach to multi-path programs is also part of our future work.



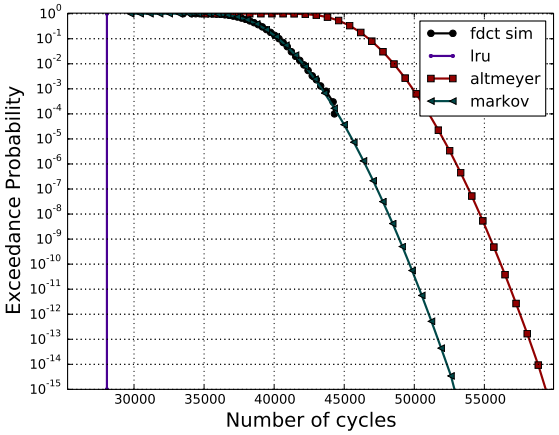
(a) Cache size: 256 bytes. Associativity: 2.



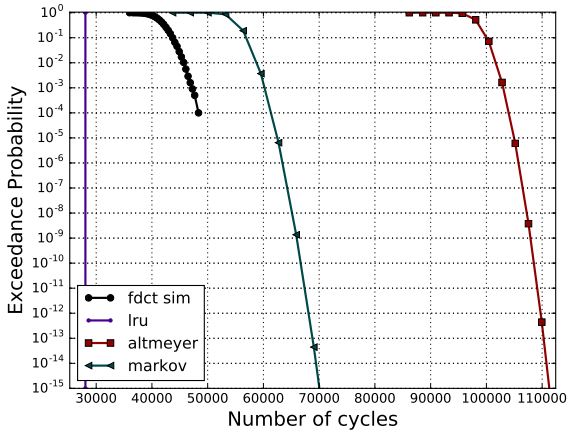
(b) Cache size: 256 bytes. Associativity: 4.



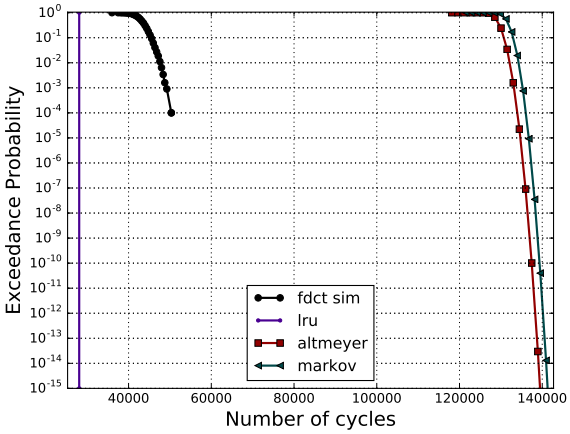
(c) Cache size: 512 bytes. Associativity: 2.



(d) Cache size: 512 bytes. Associativity: 4.



(e) Cache size: 512 bytes. Associativity: 8.



(f) Cache size: 512 bytes. Associativity: 16.

Figure 7.9 Benchmark fdct. Comparison with LRU replacement with different cache sizes and associativities. Number of used blocks n=6.

CHAPTER 8 GENERAL DISCUSSION

In previous chapters, we have presented the details of our research work. In this chapter, we discuss their contributions and potential impact in three aspects: timing analysis and complexity, impact of faults and the integration of fault detection. At the end of the chapter, we summarize the contributions and the impact in Table 8.1.

8.1 On Timing Analysis Accuracy and Complexity

The use of random caches has become increasingly popular for embedded systems, which makes it necessary to develop corresponding timing analysis techniques. As a result, probabilistic timing analysis methods have been proposed to compute timing distributions with respect to exceedance probabilities. In Chapters 3 and 7, we have proposed a Markov chain based SPTA to calculate timing estimates accurately.

In Chapter 3, we aimed at developing an SPTA method in the presence of faults. Using a Markov chain model, we are able to describe all states of the system at each cache access. Thus, the timing information could be precisely obtained. Nevertheless, there exists a challenge in using the Markov chain model: the number of states increases so dramatically with the number of distinct memory addresses that it becomes intractable. Consequently, we limited the number of states to make the SPTA tractable.

In Chapter 7, we saw that the state number of Markov chain based SPTA increases polynomially with a large exponent. Besides, the prior research work focused only on instruction caches. We extended the work in Chapter 3 from instruction caches to write-back data caches with a *write-allocate* policy, which is a commonly used policy to reduce data access latencies (the SPTA can be extended to write-through data caches as well with trivial modifications). The adaptive state modification method, which obtains timing information with limited number of states, is explained in detail to demonstrate its safety and tightness.

Experiments in Chapter 3 illustrate that the Markov chain based SPTA can always produce safe pWCET estimates, and accurate pWCET estimates can be calculated with enough number of memory addresses. Although we observed some differences at low exceedance probabilities between the simulations and the pWCET distributions obtained using the proposed SPTA, this can be explained by the insufficient number of data-points retrieved from the simulations. Experiments in Chapter 7 show that, with extension to data caches, the Markov chain based SPTA can still provide accurate results. Statistically, our proposed method

compute more accurate pWCET estimates compared to the state-of-the-art combined SPTA method.

Using our proposed SPTA method, we can obtain pWCET estimates of random instruction and data caches, which allows further analysis of the timing behavior of the system. For example, we can compare pWCET estimates of random caches with timing estimates from LRU caches, as performed in Chapter 7. We discover that although in many cases LRU caches perform better, they also perform very poorly in some other cases, i.e. the execution times using LRU caches are much longer than those using random caches. This is due to the occurrence of systematic cache misses in LRU caches, especially in loops where the number of accessed memory blocks is larger than the cache associativity. Random caches avoid this phenomenon by evicting cache blocks randomly, which helps improve the WCET of a system. This confirms conclusions from previous studies [90, 27]. In terms of average performance, it is hard to extract absolute conclusions on the best choice between random and LRU caches. Smith and Goodman [100, 101] argue that random replacement outperforms LRU replacement. However, Quinones *et al.* [90] suggest that random replacement can degrade average performance. Since we could not obtain all possible memory traces of the benchmarks, we did not evaluate the average performance of random replacement.

Although our SPTA method is proposed for random caches, it can also be applied to deterministic caches (e.g. LRU caches), in which each memory access can be regarded as a special case of random caches with a deterministic probability. However, in terms of performance, it may not be comparable to state-of-the-art timing analysis method for deterministic caches, since it was initially proposed for random caches.

8.2 On the Impact of Transient and Permanent Faults

As feature sizes of semiconductor devices scale down, reliability issues have arisen: the fault probabilities of caches–the components that adopt smallest features allowed and occupy large area of processors–increases steadily. This lead us to the development of SPTA method in the presence of faults in Chapters 3 and 5.

In this dissertation, we focus on faults in storage elements of caches. Faults in the combinational logic are different from those in the storage elements. For example, transient faults in the combinational logic do not affect future operations, while those in the storage elements will not be removed until they are detected. To prevent errors in the combinational logic, corresponding fault detection and correction techniques can be adopted, which may increase time to access the cache. This is however out of the scope of the dissertation.

We have modelled both transient and permanent faults for random caches in Chapter 3. For transient faults, we focus on SEUs (a change of state by high-energy particles, especially in space environments) which produce errors temporarily. Cache blocks with transient faults provide erroneous data. Upon detecting transient faults, the system sets the data in the corresponding block as invalid and may fill the cache blocks with new memory data. Therefore faults are removed and the cache blocks can work without affecting future operations.

There are different types of permanent faults due to process variations in manufacturing process, aging effects of components, etc. We study dynamic permanent faults induced by aging effects. By establishing a formula to calculate the permanent fault probability, we observe that the permanent fault probability increases with time. Once a permanent fault occurs, it remains in the cache block. To avoid following errors, this cache block can be disabled for further use.

With fault models, we successfully analyze fault impact on pWCET estimates. Chapter 3 presents our first attempt to statically analyze random caches in presence of faults. When transient fault rate increases, the pWCET estimate increases gradually as what we would have expected. We note that, however, as the number of permanent fault exceeds some threshold, there may be a dramatic increase in execution time for some benchmarks. This is in reason of the fact that, after a permanent fault occurs in a cache set, the program may frequently access that same set, which significantly increases the execution times due to cache misses caused by permanent faults.

Chapter 5 extends the SPTA from instruction caches to write-through data caches with *no write-allocate* policy, which avoid data inconsistency issues that happen in write-back data caches because of faults. Evaluations reveal that as the cache block size increases, the pWCET may be improved at low exceedance probabilities, but it becomes worse at high exceedance probabilities. This is because every time a cache miss occurs, more memory blocks are fetched and stored in the cache, which effectively reduces the probability of cache misses for the following memory accesses, which improves pWCET at low exceedance probabilities. Nevertheless, a permanent fault in a cache block of a large size affects more area, which degrades the pWCET at high exceedance probabilities.

From Chapters 3 and 5, we conclude that the final pWCET depends on the characteristics of the selected benchmark. To predict fault impact, we need to use our proposed SPTA and pay attention to sudden changes of pWCET estimates, in which the presence of permanent faults may vary the pWCET significantly.

8.3 On the Integration of Fault Detection into Timing Analysis

In our previous discussion, we have investigated SPTA methods that includes both transient and permanent fault impact. For simplicity reasons, a perfect fault detection technique that detects transient and permanent faults immediately after they occur is assumed to be in place.

In practice, however, a perfect detection technique is infeasible and may take some time to detect and classify a permanent fault. To improve system performance, we apply D-HMM based permanent fault detection to random caches. Through the comparison between rule-based and D-HMM based permanent fault detection in Chapter 4, we observe that fault detection plays an important role in the pWCET distributions, especially under extreme conditions when fault rates are particularly high. D-HMM predicts the occurrence of a permanent fault more accurately, which can further help disable the corresponding cache block to avoid future cache misses caused by this block. By comparing the measured execution times using rule-based and D-HMM based detection to those using a perfect detection, we discover that D-HMM detection performs very well with similar execution times to those with a perfect detection.

Having recognized the importance of permanent fault detection techniques, we integrated the implementation a practical permanent fault detection technique into the proposed SPTA in Chapter 6. Experimental evaluations show that it is possible to obtain accurate pWCETs on a system with a realistic permanent fault detection technique. Besides, we observe that when we increase the size of the cache block, the number of cache misses decreases drastically and the pWCETs are improved. We find that when we increase the cache associativity and use the same number of cache sets, the pWCETs are improved as well. In some cases, the pWCET improvements are greater when increasing the cache block size rather than increasing the cache associativity. In other cases, the pWCET improvements are smaller. This degree of improvement depends on the program features.

The integration of the rule-based detection into SPTA is the starting point of a realistic permanent detection technique analysis. More advanced detection techniques (e.g. D-HMM based detection) can be studied in a similar manner using different detection models. With fault models and detection techniques, the method can be used for aerospace application analysis, where faults occur due to high space radiation. The developed technology can be used for software validation and certification to reduce the cost significantly.

Table 8.1 Summary of the contributions and impact of the dissertation.

Ch.	Ref.	Contribution	Impact
3	[31]	Fault models for transient faults (i.e. SEUs) and dynamic permanent faults due to aging effects.	Providing researchers with fault models for caches.
		Transient and permanent fault detection techniques.	Allowing researchers to study system timing behavior with fault detection techniques.
		A Markov chain model for SPTA of random caches.	Enabling accurate timing distributions for systems in presence of faults, taking fault detection effects into consideration.
4	[30]	Proposal of D-HMM based detection for permanent faults in random caches.	Demonstrating how to implement a practical permanent fault detection technique.
		Comparison using measurements between two practical permanent fault detection techniques.	Discovery on the significance of a permanent fault detection technique on the system performance.
5	[32]	In addition to instruction caches, write-through data caches are taken into account for SPTA with faults.	Allowing researchers to study both instruction and data cache effects.
		Introduction of a cache contention based SPTA method.	Making it faster to perform SPTA at the expense of calculation speed.
		Practical fault detection comparison to perfect detection with instruction and write-through data caches.	Evaluating practical detection performance with respect to the perfect one.
6	[34]	Development of SPTA for a practical permanent fault detection technique.	Insight into integration of periodic fault detection in practice.
		Experiments with different cache configurations.	Unveiling cache configuration impact on timing distributions using the SPTA.
7	[33]	Extension of the Markov chain based SPTA to commonly used write-back data caches.	Enabling full performance analysis on both instruction and data caches.
		Performance comparison between random caches and LRU caches.	Indicating advantages and disadvantages of random caches.

CHAPTER 9 CONCLUSION AND FUTURE WORK

9.1 Contributions

In this dissertation, we have developed a Markov chain based SPTA. The state is used to represent the memory layout of the cache. For each memory access, the state update is computed using a transition matrix, which depends on the current state and the incoming memory block. The timing information for each state is calculated separately, and the pWCET distribution can be derived using the timing information of all states after accessing all memory blocks. We limit the number of memory blocks in the Markov chain model and use an adaptive method to change the memory blocks in the state and to retain as much information as possible. By comparing the proposed SPTA method to state-of-the-art SPTA method with a combination of a state enumeration method and a cache contention method, we discover that on average, the Markov chain based SPTA generates a more precise pWCET estimate.

To deal with the impact of faults, we have defined a transient fault model and a permanent fault model. The transient fault model considers SEUs as transient faults, and we assume that an online detection technique can detect all transient faults. The permanent fault model assumes that for every memory access, there is a probability of a permanent error in the program execution caused by the system's components wear-out, the aging effects, etc. This probability is not constant and it increases as time goes by. We suppose that there is a permanent fault detection mechanism to detect and classify permanent faults. Once a permanent fault is detected, we disable the cache block where it occurred so that only fault-free blocks are used for future memory accesses.

With the fault and detection techniques in place, we have integrated the fault impacts into our Markov chain based SPTA and verified its accuracy through a series of experiments. Since our SPTA is based on states, we manage to incorporate the fault impact into the state enumeration method of the state-of-the-art SPTA. We develop the formulae to account for the impact of faults on cache contention method separately. The experimental results prove the effectiveness of the modified state-of-the-art SPTA that takes faults into consideration.

There are many ways to implement a permanent fault detection in real systems. In this dissertation, we deploy two permanent fault detection techniques—rule-based and D-HMM based detection—and study the impact of the implementation of permanent fault detection using measurements. We conclude that, on average, permanent fault detection provides

execution times that are shorter compared to those using rule-based detection.

We then integrate the rule-based permanent fault detection into a state-of-the-art SPTA method to obtain pWCET statically. Since the permanent fault detection technique is periodic, we use two modes to represent the case in which the fault detection is active and the case in which the fault detection is idle. For each mode, we develop corresponding formulae to provide safe and tight pWCETs. In the experiment section, we evaluate the accuracy and impact of faults on pWCET distributions. The role of the cache block size and cache associativity are investigated as well.

9.2 Limitations

We have proposed a number of SPTA methodologies able to cope with the presence of faults. However, there are some constraints to their application. As a baseline for the study of the implementation impacts on pWCET distributions (from SPTA) we use a rule-based permanent fault detection. This is a very simple fault detection mechanism, and there are other advanced mechanisms, such as D-HMM based detection, which produce improved pWCET estimates. Our SPTA only applies only to rule-based detection and, to integrate other detection mechanisms, additional formulae should be developed, a non-trivial task. Furthermore, when permanent fault detection is implemented, transient faults are absent. However, their occurrences may affect the effectiveness of the rule-based detection, which requires modifications of our proposed SPTA methodology.

A second limitation is the applicability of our SPTA in the presence of faults. We have developed our SPTA for single-path programs, and added fault impacts in the SPTA for cases in the presence of faults. However, our SPTA does not apply to multi-path programs. Note that, when we verify its accuracy, we compare the pWCET distributions with those derived from simulations rather than using industrial measurements.

9.3 Future Work

For our future work, we will extend our single-path SPTA with faults to multi-path SPTA. Furthermore, we used the Mälardalen benchmark suit in our experiments, a common suit for PTA evaluations. However, there are other available benchmarks and we plan on evaluating our methods using them as well. Since there are not many commercial products with random caches, we have only verified pWCET distributions using simulations. It would be interesting for us to perform SPTA on real commercial embedded-systems in future.

REFERENCES

[1] "IEC 61508:2010 CMV: Functional safety of electrical /electronic /programmable electronic safety-related system," 2010. [Online]. Available: http://www.iec.ch/functionalsafety/

[2] J. Abella, P. Chaparro, X. Vera, J. Carretero, and A. Gonzalez, "On-line failure detection and confinement in caches," in *On-Line Testing Symposium, 2008. IOLTS '08. 14th IEEE International*, July 2008, pp. 3–9.

[3] J. Abella, P. Chaparro, X. Vera, J. Carretero, and A. González, "On-line failure detection and confinement in caches," in *2008 14th IEEE International On-Line Testing Symposium*, July 2008, pp. 3–9.

[4] J. Abella, D. Hardy, I. Puaut, E. Quinones, and F. Cazorla, "On the comparison of deterministic and probabilistic wcet estimation techniques," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, July 2014, pp. 266–275.

[5] J. Abella, E. Quinones, F. Wartel, T. Vardanega, and F. Cazorla, "Heart of gold: Making the improbable happen to increase confidence in mbpta," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, July 2014, pp. 255–265.

[6] J. Abella, C. Hernandez, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega, "Wcet analysis methods: Pitfalls and challenges on their trustworthiness," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, June 2015, pp. 1–10.

[7] I. Agirre, M. Azkarate-askasua, C. Hernandez, J. Abella, J. Perez, T. Vardanega, and F. J. Cazorla, "Iec-61508 sil 3 compliant pseudo-random number generators for probabilistic timing analysis," in *2015 Euromicro Conference on Digital System Design*, Aug 2015, pp. 677–684.

[8] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite," in *Proceedings of the 42Nd Annual Southeast Regional Conference*, ser. ACM-SE 42. New York, NY, USA: ACM, 2004, pp. 267–272.

[9] S. Altmeyer and R. Davis, "On the correctness, optimality and precision of static probabilistic timing analysis," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, pp. 1–6.

[10] S. Altmeyer, L. Cucu-Grosjean, and R. I. Davis, "Static probabilistic timing analysis for real-time systems using random replacement caches," *Real-Time Syst.*, vol. 51, no. 1, pp. 77–123, Jan. 2015.

[11] S. Altmeyer, L. Cucu-Grosjean, and R. Davis, "Static probabilistic timing analysis for real-time systems using random replacement caches," *Real-Time Systems*, vol. 51, no. 1, pp. 77–123, 2015.

[12] S. I. Association *et al.*, "International technology roadmap for semiconductors, 2009 edition," *International SEMATECH, Austin, Texas*, 2009.

[13] J. Beirlant, Y. Goegebeur, J. Segers, and J. Teugels, *Statistics of Extremes: Theory and Applications*. John Wiley & Sons, 2006.

[14] P. Benedicte, L. Kosmidis, E. Quinones, J. Abella, and F. J. Cazorla, "Modelling the confidence of timing analysis for time randomised caches," in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016, pp. 1–8.

[15] K. Berezovskyi, F. Guet, L. Santinelli, K. Bletsas, and E. Tovar, "Measurement-based probabilistic timing analysis for graphics processor units," in *Architecture of Computing Systems – ARCS 2016: 29th International Conference, Nuremberg, Germany, April 4–7, 2016, Proceedings*, F. Hannig, J. M. P. Cardoso, T. Pionteck, D. Fey, W. Schröder-Preikschat, and J. Teich, Eds. Cham: Springer International Publishing, 2016, pp. 223–236.

[16] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," *SIGPLAN Not.*, vol. 41, no. 6, pp. 158–168, Jun. 2006.

[17] P. Bernardara, F. Mazas, X. Kergadallan, and L. Hamm, "A two-step framework for over-threshold modelling of environmental extremes," *Natural Hazards and Earth System Science*, vol. 14, no. 3, pp. 635–647, 2014.

[18] G. Bernat, A. Colin, and S. Petters, "Wcet analysis of probabilistic hard real-time systems," in *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, 2002, pp. 279–288.

[19] G. Bernat, A. Colin, J. Esteves, G. Garcia, C. Moreno, N. Holsti, T. Vardanega, and M. Hernek, "Considerations on the leon cache effects on the timing analysis of on-board applications," in *Proceedings of the Data Systems in Aerospace Conference (DASIA)*, 2007.

[20] G. Bernat, A. Colin, and S. Petters, "pwcet: A tool for probabilistic worst-case execution time analysis of real-time systems," *Technical report YCS-353-2003, Department of Comptuer Science, University of York*, 2003.

[21] G. Bernat, A. Burns, and M. Newby, "Probabilistic timing analysis: An approach using copulas," *J. Embedded Comput.*, vol. 1, no. 2, pp. 179–194, Apr. 2005.

[22] A. Betts, N. Merriam, and G. Bernat, "Hybrid measurement-based WCET analysis at the source level using object-level traces," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, ser. OpenAccess Series in Informatics (OASIcs), B. Lisper, Ed., vol. 15. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 54–63, the printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.

[23] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[24] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proceedings of the 40th Annual Design Automation Conference*, ser. DAC '03. New York, NY, USA: ACM, 2003, pp. 338–342.

[25] K. Bowman, J. Tschanz, C. Wilkerson, S.-L. Lu, T. Karnik, V. De, and S. Borkar, "Circuit techniques for dynamic variation tolerance," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. New York, NY, USA: ACM, 2009, pp. 4–7.

[26] A. Burns and S. Edgar, "Predicting computation time for advanced processor architectures," in *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, 2000, pp. 89–96.

[27] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim, "Proartis: Probabilistically analyzable real-time systems," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, pp. 94:1–94:26, May 2013.

[28] F. J. Cazorla, T. Vardanega, E. Quiñones, and J. Abella, "Upper-bounding Program Execution Time with Extreme Value Theory," in *13th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), C. Maiza, Ed., vol. 30. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 64–76.

[29] S. Chattopadhyay and A. Roychoudhury, "Unified cache modeling for wcet analysis and layout optimizations," in *2009 30th IEEE Real-Time Systems Symposium*, Dec 2009, pp. 47–56.

[30] C. Chen, J. Panerati, and G. Beltrame, "Effects of online fault detection mechanisms on probabilistic timing analysis," in *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Sept 2016, pp. 41–46.

[31] C. Chen, L. Santinelli, J. Hugues, and G. Beltrame, "Static probabilistic timing analysis in presence of faults," in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016, pp. 1–10.

[32] C. Chen, J. Panerati, and G. Beltrame, "Probabilistic timing analysis of random caches with fault detection mechanisms," *IEEE Transactions on Emerging Topics in Computing*, vol. 3, Sept 2017, submitted.

[33] C. Chen and G. Beltrame, "An adaptive markov model for the timing analysis of probabilistic caches," *ACM Trans. Des. Autom. Electron. Syst.*, 2017, accepted.

[34] C. Chen, J. Panerati, I. Hafnaoui, and G. Beltrame, "Static probabilistic timing analysis with a permanent fault detection mechanism," in *2017 12th IEEE Symposium on Industrial Embedded Systems (SIES)*, June 2017, accepted.

[35] A. Colin and I. Puaut, "Worst-case execution time analysis of the rtems real-time operating system," in *Proceedings 13th Euromicro Conference on Real-Time Systems*, 2001, pp. 191–198.

[36] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, Jul. 2003.

[37] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, July 2012, pp. 91–101.

[38] C. Curtsinger and E. D. Berger, "Stabilizer: Statistically sound performance evaluation," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 219–228, Mar. 2013.

[39] R. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean, "Analysis of probabilistic cache related pre-emption delays," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, July 2013, pp. 168–179.

[40] R. Davis, "Improvements to static probabilistic timing analysis for systems with random cache replacement policies," *RTSOPS 2013*, pp. 22–24, 2013.

[41] L. De Haan and A. Ferreira, *Extreme value theory: an introduction*. Springer, 2007.

[42] E. Díaz, J. Abella, E. Mezzetti, I. Agirre, M. Azkarate-Askasua, T. Vardanega, and F. J. Cazorla, "Mitigating Software-Instrumentation Cache Effects in Measurement-Based Timing Analysis," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 1–11.

[43] B. Dreyer, C. Hochberger, A. Lange, S. Wegener, and A. Weiss, "Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 1–11.

[44] S. Edgar and A. Burns, "Statistical analysis of wcet for scheduling," in *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, Dec 2001, pp. 215–224.

[45] E. A. Elsayed, *Reliability engineering*. John Wiley & Sons, 2012.

[46] ESA, "System and Technology Study Report, Chap4: The Mercury Environment," April 2000.

[47] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, *Reliable and Precise WCET Determination for a Real-Life Processor*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 469–485.

[48] D. Griffin and A. Burns, "Realism in statistical analysis of worst case execution times." in *WCET*, 2010, pp. 44–53.

[49] D. Griffin, B. Lesage, A. Burns, and R. I. Davis, "Static probabilistic timing analysis of random replacement caches using lossy compression," in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. New York, NY, USA: ACM, 2014, pp. 289:289–289:298.

[50] S. Guertin and M. White, "Cmos reliability challenges the future of commercial digital electronics and nasa," in *NEPP Electronic Technology Workshop*, 2010.

[51] E. J. Gumbel, *Statistics of extremes*. Courier Corporation, 2012.

[52] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks: Past, Present And Future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, ser. OpenAccess Series in Informatics (OASIcs), vol. 15, 2010, pp. 136–146.

[53] J. Hansen, S. A. Hissam, and G. A. Moreno, "Statistical-based wcet estimation and validation," in *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[54] D. Hardy and I. Puaut, "Wcet analysis of multi-level non-inclusive set-associative instruction caches," in *2008 Real-Time Systems Symposium*, Nov 2008, pp. 456–466.

[55] D. Hardy, I. Puaut, and Y. Sazeides, "Probabilistic wcet estimation in presence of hardware for mitigating the impact of permanent faults," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 91–96.

[56] D. Hardy and I. Puaut, "Static probabilistic worst case execution time estimation for architectures with faulty instruction caches," in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, ser. RTNS '13. New York, NY, USA: ACM, 2013, pp. 35–44.

[57] D. Hardy and I. Puaut, "Wcet analysis of instruction cache hierarchies," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 677 – 694, 2011, special Issue on Worst-Case Execution-Time Analysis.

[58] D. Hardy, I. Sideris, N. Ladas, and Y. Sazeides, "The performance vulnerability of architectural and non-architectural arrays to permanent faults," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 48–59.

[59] C. Hernandez, J. Abella, F. J. Cazorla, J. Andersson, and A. Gianarro, "Towards making a leon3 multicore compatible with probabilistic timing analysis," in *DASIA*, 2015.

[60] C. Hernandez, J. Abella, A. Gianarro, J. Andersson, and F. J. Cazorla, "Random modulo: A new processor cache design for real-time critical systems," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 29:1–29:6.

[61] J. Jalle, L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla, "Bus designs for time-probabilistic multicore processors," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '14. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014, pp. 50:1–50:6.

[62] L. Kosmidis, J. Abella, F. Wartel, E. Quiñones, A. Colin, and F. J. Cazorla, "Pub: Path upper-bounding for measurement-based probabilistic timing analysis," in *2014 26th Euromicro Conference on Real-Time Systems*, July 2014, pp. 276–287.

[63] L. Kosmidis, E. Quinones, J. Abella, T. Vardanega, I. Broster, and F. Cazorla, "Measurement-based probabilistic timing analysis and its impact on processor architecture," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*, Aug 2014, pp. 401–410.

[64] L. Kosmidis, E. Quiñones, J. Abella, G. Farrall, F. Wartel, and F. J. Cazorla, "Containing timing-related certification cost in automotive systems deploying complex hardware," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2014, pp. 1–6.

[65] L. Kosmidis, R. Vargas, D. Morales, E. Quiñones, J. Abella, and F. J. Cazorla, "Tasa: Toolchain-agnostic static software randomisation for critical real-time systems," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2016, pp. 1–8.

[66] L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla, "A cache design for probabilistically analysable real-time systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. San Jose, CA, USA: EDA Consortium, 2013, pp. 513–518.

[67] L. Kosmidis, C. Curtsinger, E. Quiñones, J. Abella, E. Berger, and F. J. Cazorla, "Probabilistic timing analysis on conventional cache designs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. San Jose, CA, USA: EDA Consortium, 2013, pp. 603–606.

[68] L. Kosmidis, T. Vardanega, J. Abella, E. Quiñones, and F. J. Cazorla, "Applying Measurement-Based Probabilistic Timing Analysis to Buffer Resources," in *13th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), C. Maiza, Ed., vol. 30. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 97–108.

[69] L. Kosmidis, E. Quiñones, J. Abella, T. Vardanega, C. Hernandez, A. Gianarro, I. Broster, and F. J. Cazorla, "Fitting processor architectures for measurement-based probabilistic timing analysis," *Microprocessors and Microsystems*, vol. 47, Part B, pp. 287 – 302, 2016.

[70] K. Lahiri, A. Raghunathan, and G. Lakshminarayana, "Lotterybus: A new high-performance communication architecture for system-on-chip designs," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: ACM, 2001, pp. 15–20.

[71] B. Lesage, D. Griffin, S. Altmeyer, and R. Davis, "Static probabilistic timing analysis for multi-path programs," in *Real-Time Systems Symposium, 2015 IEEE*, Dec 2015, pp. 361–372.

[72] B. Lesage, D. Griffin, F. Soboczenski, I. Bate, and R. I. Davis, "A framework for the evaluation of measurement-based timing analyses," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, ser. RTNS '15. New York, NY, USA: ACM, 2015, pp. 35–44.

[73] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, no. 1, pp. 56 – 67, 2007, special issue on Experimental Software and Toolkits.

[74] Y.-T. S. Li and S. Malik, *Performance analysis of real-time embedded software*. Springer Science & Business Media, 2012.

[75] Y. Lu, T. Nolte, I. Bate, and L. Cucu-Grosjean, "A new way about using statistical analysis of worst-case execution times," *SIGBED Rev.*, vol. 8, no. 3, pp. 11–14, Sep. 2011.

[76] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A survey on static cache analysis for real-time systems," *Leibniz Transactions on Embedded Systems*, vol. 3, no. 1, pp. 05–1–05:48, 2016.

[77] G. Martin, "Overview of the mpsoc design challenge," in *Proceedings of the 43rd Annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: ACM, 2006, pp. 274–279.

[78] E. Mezzetti, M. Ziccardi, T. Vardanega, J. Abella, E. Quiñones, and F. J. Cazorla, "Randomized caches can be pretty useful to hard real-time systems," *Leibniz Transactions on Embedded Systems*, vol. 2, no. 1, pp. 01–1, 2015.

[79] K. Mohr and L. Clark, "Delay and area efficient first-level cache soft error detection and correction," in *Computer Design, 2006. ICCD 2006. International Conference on*, Oct 2006, pp. 88–92.

[80] F. Mueller, "Timing analysis for instruction caches," *Real-Time Systems*, vol. 18, no. 2, pp. 217–247, 2000.

[81] B. Muirhead and L. Fesq, "Managing space system faults: Coalescing nasa's views," in *2012 IEEE Aerospace Conference*, March 2012, pp. 1–8.

[82] S. R. Nassif, N. Mehta, and Y. Cao, "A resilience roadmap," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 1011–1016.

[83] S. R. Nassif, N. Mehta, and Y. Cao, "A resilience roadmap," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 1011–1016.

[84] H. S. Negi, T. Mitra, and A. Roychoudhury, "Accurate estimation of cache-related preemption delay," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '03. New York, NY, USA: ACM, 2003, pp. 201–206.

[85] E. Normand, "Single-event effects in avionics," *Nuclear Science, IEEE Transactions on*, vol. 43, no. 2, pp. 461–474, Apr 1996.

[86] J. Panerati, S. Abdi, and G. Beltrame, "Balancing system availability and lifetime with dynamic hidden markov models," in *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, July 2014, pp. 240–247.

[87] M. Panic, J. Abella, C. Hernandez, E. Quiñones, T. Ungerer, and F. J. Cazorla, "Enabling tdma arbitration in the context of mbpta," in *2015 Euromicro Conference on Digital System Design*, Aug 2015, pp. 462–469.

[88] S. M. Petters, "Comparison of trace generation methods for measurement based wcet analysis," in *In Proceedings of the 3rd International Workshop on Worst Case Execution Time Analysis*, 2003, pp. 61–64.

[89] I. Puaut, "Wcet-centric software-controlled instruction caches for hard real-time systems," in *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, 2006, pp. 10 pp.–226.

[90] E. Quinones, E. Berger, G. Bernat, and F. Cazorla, "Using randomized caches in probabilistic real-time systems," in *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, July 2009, pp. 129–138.

[91] H. Ramaprasad and F. Mueller, "Bounding worst-case data cache behavior by analytically deriving cache reference patterns," in *11th IEEE Real Time and Embedded Technology and Applications Symposium*, March 2005, pp. 148–157.

[92] J. Reineke, "Randomized caches considered harmful in hard real-time systems," *Leibniz Transactions on Embedded Systems*, vol. 1, no. 1, pp. 03–1–03:13, 2014.

[93] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Systems*, vol. 37, no. 2, pp. 99–122, 2007.

[94] P. Ryan Conmy, M. Pearce, M. Ziccardi, E. Mezzetti, T. Vardanega, J. Anderson, A. Gianarro, C. Hernandez, and F. J. Cazorla, "Measurement-Based Probabilistic Timing Analysis - From Academia to Space Industry," in *DASIA 2015 - DAta Systems in Aerospace*, ser. ESA Special Publication, vol. 732, Sep. 2015, p. 61.

[95] R. Schaefer, "Unmanned aerial vehicle reliability study," *Office of the Secretary of Defense, Washington, DC*, 2003.

[96] M. Schlansker, R. Shaw, and S. Sivaramakrishnan, *Randomization and associativity in the design of placement-insensitive caches*. Hewlett-Packard Laboratories, 1993.

[97] R. Serfozo, *Basics of applied stochastic processes*. Springer, 2009.

[98] M. Slijepcevic, L. Kosmidis, J. Abella, E. Quinones, and F. Cazorla, "Dtm: Degraded test mode for fault-aware probabilistic timing analysis," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, July 2013, pp. 237–248.

[99] M. Slijepcevic, L. Kosmidis, J. Abella, E. Quinones, and F. Cazorla, "Timing verification of fault-tolerant chips for safety-critical applications in harsh environments," *Micro, IEEE*, vol. 34, no. 6, pp. 8–19, Nov 2014.

[100] J. E. Smith and J. R. Goodman, "A study of instruction cache organizations and replacement policies," in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ser. ISCA '83. New York, NY, USA: ACM, 1983, pp. 132–137.

[101] J. E. Smith and J. R. Goodman, "Instruction cache replacement policies and organizations," *IEEE Transactions on Computers*, vol. 34, no. 3, pp. 234–241, 1985.

[102] SPEC2000, "Standard Performance Evaluation Corporatio."

[103] J. Staschulat, S. Schliecker, and R. Ernst, "Scheduling analysis of real-time systems with precise modeling of cache related preemption delay," in *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, July 2005, pp. 41–48.

[104] Z. Stephenson, J. Abella, and T. Vardanega, "Supporting industrial use of probabilistic timing analysis with explicit argumentation," in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, July 2013, pp. 734–740.

[105] A. Taber and E. Normand, "Single event upset in avionics," *IEEE Transactions on Nuclear Science*, vol. 40, no. 2, pp. 120–126, Apr 1993.

[106] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise wcet prediction by separated cache andpath analyses," *Real-Time Syst.*, vol. 18, no. 2/3, pp. 157–179, May 2000.

[107] N. Topham and A. Gonzalez, "Randomized cache placement for eliminating conflicts," *Computers, IEEE Transactions on*, vol. 48, no. 2, pp. 185–192, Feb 1999.

[108] L. Trichtchenko, L. Nikitina, A. Trishchenko, and L. Garand, "Highly elliptical orbits for arctic observations: Assessment of ionizing radiation," *Advances in Space Research*, vol. 54, no. 11, pp. 2398 – 2414, 2014.

[109] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Resilient random modulo cache memories for probabilistically-analyzable real-time systems," in *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, July 2016, pp. 27–32.

[110] X. Vera, B. Lisper, and J. Xue, "Data caches in multitasking hard real-time systems," in *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, Dec 2003, pp. 154–165.

[111] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quinones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega, and F. Cazorla, "Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study," in

Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on, June 2013, pp. 241–248.

[112] F. Wartel, L. Kosmidis, A. Gogonel, A. Baldovin, Z. Stephenson, B. Triquet, E. Quiñones, C. Lo, E. Mezzetti, I. Broster, J. Abella, L. Cucu-Grosjean, T. Vardanega, and F. J. Cazorla, "Timing analysis of an avionics case study on complex hardware/software platforms," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 397–402.

[113] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based worst-case execution time analysis," in *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, May 2005, pp. 7–10.

[114] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, *Measurement-Based Timing Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 430–444.

[115] R. Wilhelm, "Determining bounds on execution times." *Embedded Systems Handbook*, vol. 2, 2005.

[116] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[117] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off cache capacity for reliability to enable low voltage operation," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 2008, pp. 203–214.

[118] S. Zhou, "An efficient simulation algorithm for cache of random replacement policy," in *Network and Parallel Computing*. Springer, 2010, pp. 144–154.

[119] M. Ziccardi, E. Mezzetti, T. Vardanega, J. Abella, and F. J. Cazorla, "Epc: Extended path coverage for measurement-based probabilistic timing analysis," in *2015 IEEE Real-Time Systems Symposium*, Dec 2015, pp. 338–349.